Symmetric Multimodal Interaction

Applied to Database Design and Normalization

by

James Roy Oleinik

S.B., C.S. Massachusetts Institute of Technology, 2009

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2010

[June 2010]

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by .....
Randall Davis
Professor
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Symmetric Multimodal Interaction Applied to Database Design and Normalization

by

James Roy Oleinik

Submitted to the

Department of Electrical Engineering and Computer Science

May 21, 2010

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

A normal conversation between two people is typically multimodal, using both speech and gestures to effect communication. It is also symmetric because there is two-way multimodal interaction between the two parties. In contrast, when a human interacts with a computer, it is done through a strict and limited interface, usually a keyboard or mouse. Unlike the human-human conversation, this interaction is neither multimodal nor symmetric. The goal of this thesis is to empower computers to carry out symmetric, multimodal dialogues with humans, thereby providing a more natural human-computer interaction.

To do so, we modified and extended Adler's Multimodal Interactive DialOgue System (MIDOS) to be a more flexible and domain-independent platform for supporting symmetric, multimodal interaction. We built an application that utilizes MIDOS in order to design and implement a normalized relational database, and then demonstrate the application's capabilities by using it to design the database for an after-action report wiki.

Thesis Supervisor: Randall Davis

Title: Professor

Acknowledgements

After 5 years of long M.I.T. nights working on problems sets and research, I am finally done.

I would like to thank my academic advisor Jim Glass. Without his guidance, I would not have been able to reach the level of success I achieved in my time here at M.I.T.

I would also like to thank my thesis advisor Randall Davis. The day I first went in to talk to Professor Davis about this project, he told me straight up that it would be a lot of work, and he was right. But, ever since that day he did everything in his power to encourage me and push me in the right direction. Without his helpful suggestions and insightful questions, this thesis would not be as good as it is today. My only regret is being able to spend only one year learning from him.

Most importantly, I would like to thank my family for their constant and unconditional love and support. Thank you, Mom, Dad, and Jessica. To my parents, I would just like to say how thankful I am for all those nights when you read to me and all those times when you drove me to practice and school. You are reasons I made it into (and out of) M.I.T. and I love you both so much. Finally, I'd like to say a special thank you to my girlfriend Healy. She has been with me from the very beginning of my journey through M.I.T., and has had to endure more of my stress than anyone else. Healy, I wouldn't have made it without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Humans are able to interact with one another more easily than they can with computers. In a normal conversation between two people, the participants use multiple modes of communication, including speech and gestures, to convey information. This two-way multimodal interaction ensures that information flows quickly and effortlessly between the parties. On the other hand, when a human interacts with a computer, it is done through a strict and limited interface, usually a keyboard or mouse. The goal of this thesis is to allow for a more natural human-computer interaction (HCI) by empowering computers to carry out symmetric, multimodal dialogues with humans.

A member of our research group, Aaron Adler, created the Multimodal Interactive DialOgue System (MIDOS), which is capable of creating multimodal dialogues between a human and computer. MIDOS carries out symmetric dialogues with the user by way of speech and sketch in order to predict the behavior of these devices. The capabilities of MIDOS have already been demonstrated in the domain of simple mechanical devices.

MIDOS has established that it is possible for a computer to facilitate a symmetric, multimodal interaction with a human. However, there were several limitations with MIDOS that needed to be addressed in order to make the system a richer, more flexible framework that could be used by other programs. This thesis examines the limitations of MIDOS, discusses how we modified and extended the system to resolve these limitations, and then demonstrates the capabilities of MIDOS in the database design domain.

The first change made to MIDOS was to incorporate multiple stroke recognition, which allowed for the integration of handwriting and shape recognizers into the system. These new capabilities permitted MIDOS to support a richer sketching interface. Our next step involved making MIDOS more domain-independent, *i.e.,* capable of carrying out multimodal dialogues on a range of subjects. This required a substantial refactoring of the system in which all the domain knowledge was separated from the rest of system by the new MIDOS Interface.

We demonstrate the new capabilities of the system by using MIDOS to create an application that assists the user in the design of a relational database. First, the application

allows the user to develop an entity-relationship model by building an entity-relationship diagram. The application assists the user by engaging in a symmetric, multimodal dialogue to eliminate mistakes in the model. Next, the application translates the entity-relationship model into a relational model, and then uses the relational model to implement and run a relational database. Finally, based on the data that is stored in the relational database, the application allows the user to further refine the design of the database through a process called normalization that eliminates the storage of redundant data.

This application can assist inexperienced users in developing their own relational databases. For instance, many military bases have a need for local databases to store important information regarding details about patrols and tactics. These databases can also serve as a backend to support wikis that allow the information to be easily accessed and changed.

The application also allows the user to sketch out the design for the database, (illustrated in

Figure 1-1), and then uses MIDOS to ask the user about possible flaws in the design (Figure 1-2). With the help of the system, the user is able to easily generate a well-designed relational database like the one shown in Figure 1-3. Chapter 5 contains a complete example of the design process.

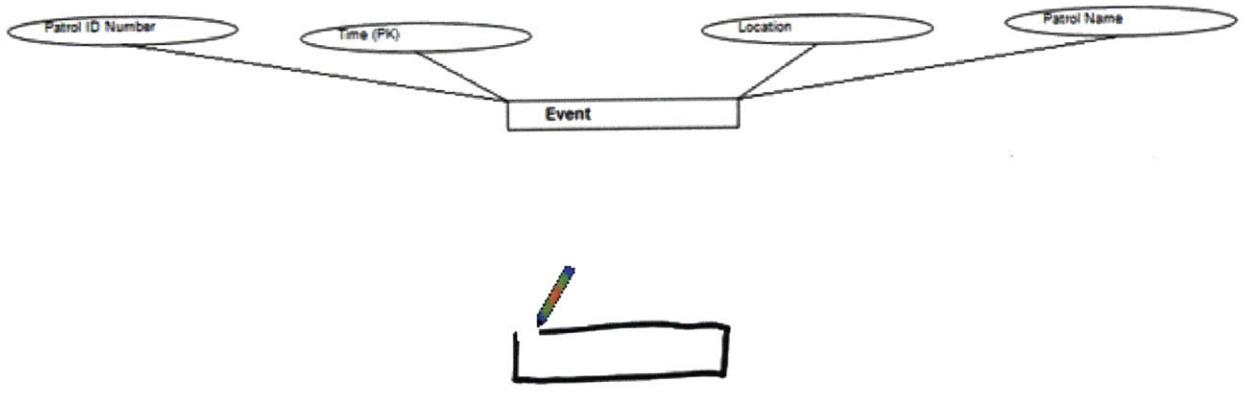


Figure 1-1: Screenshot that illustrate the user adding a new entity to the current model of the database.

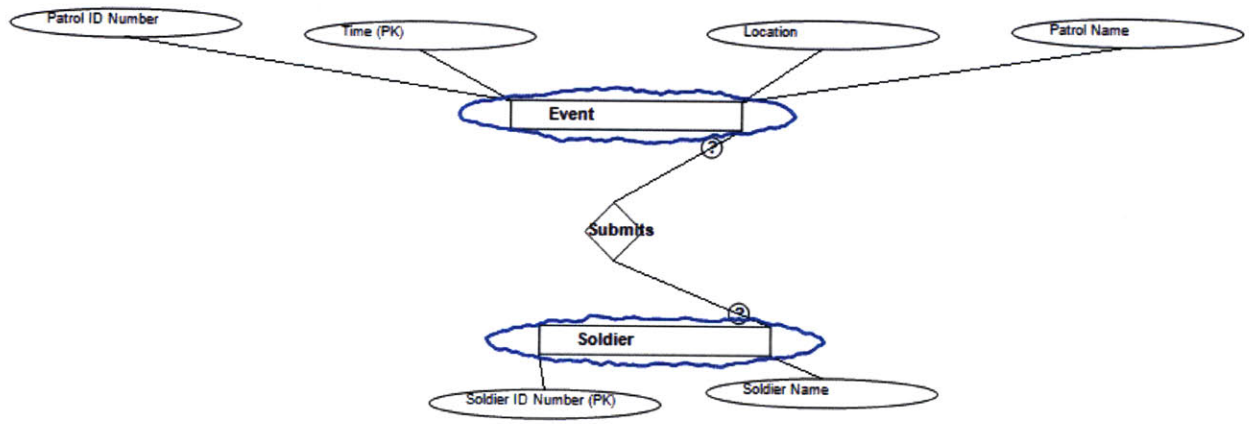


Figure 1-2: Screenshot of a sample MIDOS multimodal question where system asks the user "Can an event be associated with more than one soldier?" while identifying the Event entity and solider entity with blue circles.

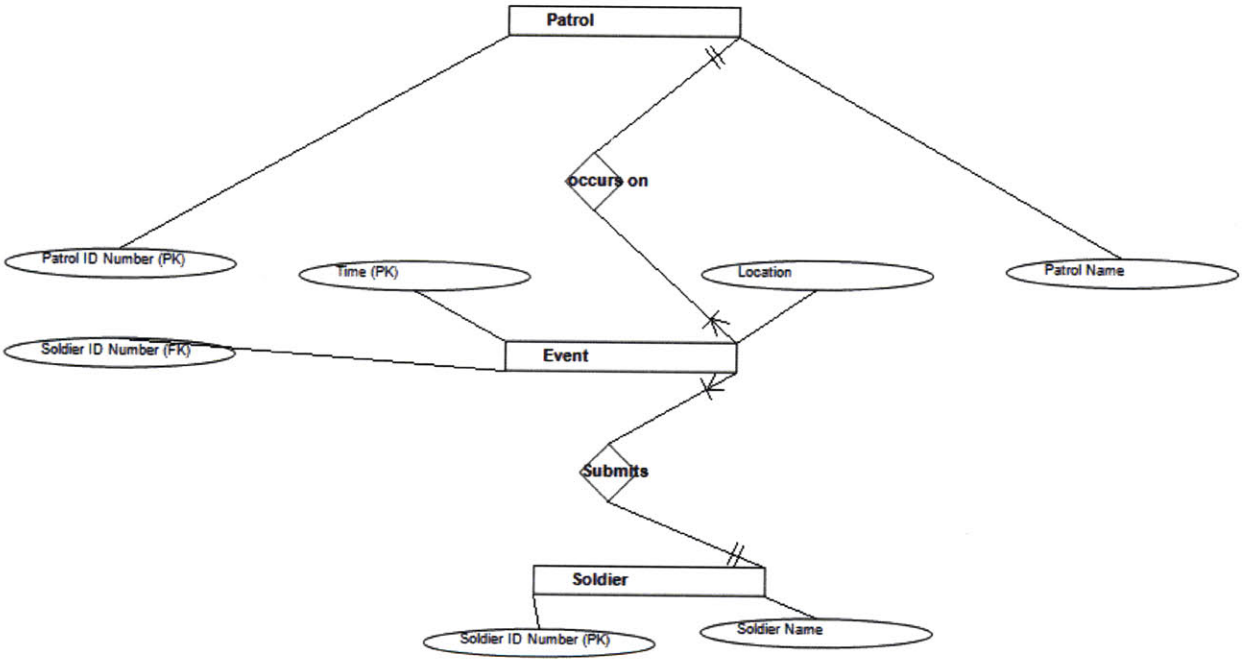


Figure 1-3: Screenshot of the final state of the database model.

The main contributions of this thesis are (1) a more flexible and domain-independent MIDOS, and (2) an application that utilizes MIDOS in order to design and implement a normalized relational database.

Chapter 2 discusses symmetric, multimodal interaction, provides an overview of MIDOS, and describes our modifications to the system in greater detail. Chapter 3 presents a general overview of the database design process. Chapter 4 discusses database normalization. Chapter 5 shows how we used MIDOS to implement an application that assists the user in the design of a database, and illustrates the capabilities of the application by providing an example of a user

designing a database for an after-action report wiki. We conclude by describing future work (Chapter 6) and our contributions (Chapter 7).

Chapter 2

Enabling Symmetric Multimodal HCI

A normal conversation between two people is typically multimodal, using both speech and gestures to effect communication, and is symmetric; there is two-way multimodal interaction between the communicating parties. Contrast that with typical HCI. A person's interaction with a computer typically involves the use of a mouse or keyboard to issue commands to the computer, which, in turn, communicates back to the user through a visual display. Unlike the human-human conversation, this interaction is neither multimodal nor symmetric. Instead of settling for this restricted form of HCI, we want to enable machines to carry out symmetric, multimodal interaction, thereby providing a more natural type of HCI.

This chapter introduces Aaron Adler's Multimodal Interactive DialOgue System (MIDOS), which is capable of creating multimodal dialogues between a human and computer in the domain of early design tasks (Adler, 2009). In particular, we provide an overview of MIDOS and discuss certain limitations of the system. We then describe how we modified and extended MIDOS to resolve these limitations.

2.1 An Overview of MIDOS

Consider the situation where a user is trying to describe the behavior of a piece of software to a computer. In order for the computer to have a good understanding of the software, the user has to expend a great deal of effort to supply the computer with every detail about the software. In the process of describing the software, the user will inevitably make mistakes, *i.e.,* contradictions and omissions, leading to an incorrect or incomplete model of the software. MIDOS was created as a way of resolving that problem and "provid[ing] an easy and natural way for the user to convey key information to the system" (Adler, 2009).

MIDOS allows the computer to use knowledge about the domain to ask intelligent questions targeted to extract specific pieces of information from the user. For example, as the user begins to describe a design, MIDOS will listen for possible omissions in the description and then ask the user clarifying questions. In this way, MIDOS allows the computer to act as a more

active participant, more closely simulating the interaction that the user would have in communicating with another person.

The original MIDOS system worked in the domain of simple mechanical devices. Each device was composed of bodies, springs, pulleys, weights, pivots and anchors. The goal of the system was to produce a simulation of the behavior of the device. In order to produce such a simulation, the system needed certain information about each object in the device. With a traditional program, if that information was missing, the program would halt or produce an error. MIDOS, by contrast, asks the user a specific question targeted to extract the unknown piece of information. The user and computer communicate with each other through a combination of speech and sketch that forms a symmetric, multimodal dialogue (Adler, 2009).

2.1.1 An Example of MIDOS

This section discusses one of the simple examples from Adler's PhD thesis illustrating the behavior of MIDOS (Adler, 2009).

Imagine that the system has been given the simple spring-block device illustrated in Figure 2-1. In this initial state, the system does not know the velocity of the block or the state of the spring. Since the spring is connected to the block, the system concludes that it can infer the velocity of the block by asking the user about the state of the spring. As a result, the system asks the user "Will this spring expand or contract" while circling the spring. In this case the user responds "It expands" while drawing a stroke from left to right. Figure 2-2 illustrates this multimodal conversation. MIDOS parses the user's response and updates the state of the device with the new velocity of the block and the state of the spring.

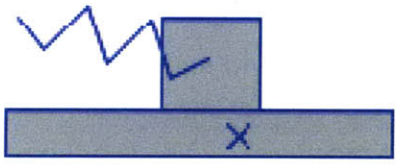


Figure 2-1: An illustration of the initial state of the spring-block mechanical device (Adler, 2009).

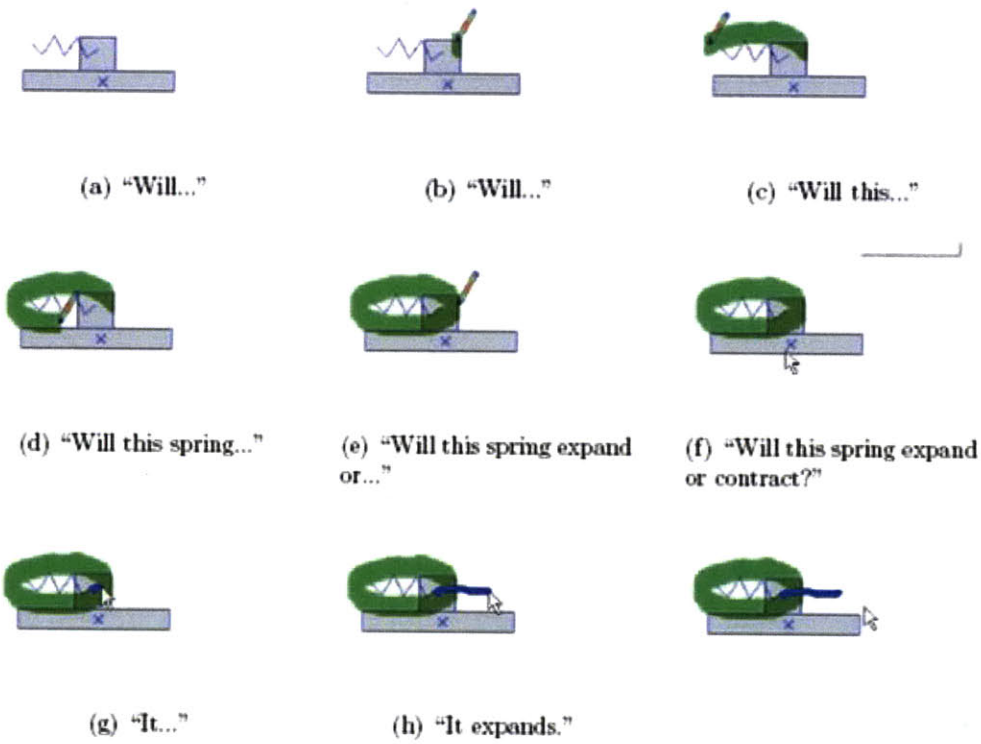


Figure 2-2: An example of MIDOS dialogue in the mechanical device domain (Adler, 2009).

2.1.2 MIDOS Architecture

MIDOS is made up of several different components: "input acquisition, output synthesis, and the core components, which include the user interface, qualitative physics simulator, question selection, and dialogue control components" (Adler, 2009). Figure 2-3 provides an illustration of these components and their interactions. The remainder of this section describes each of these components and their respective functions in MIDOS.

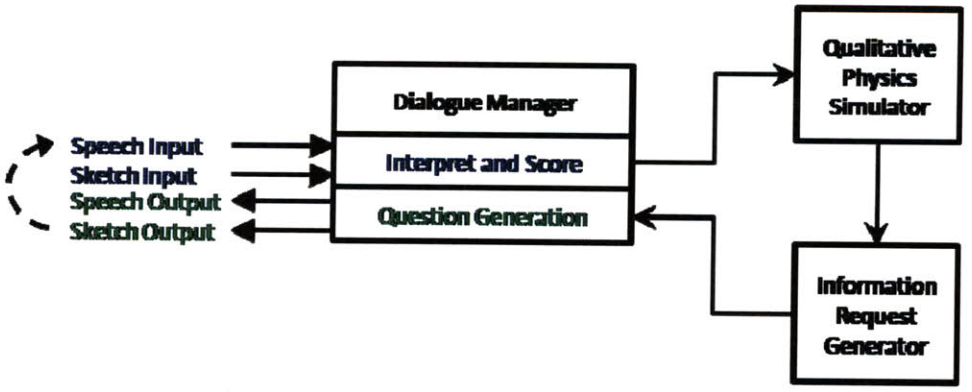


Figure 2-3: An illustration of the components and flow of information in MIDOS (Adler, 2009).

2.1.2.1 Input Acquisition and Recognition

To interpret speech input, MIDOS uses the Microsoft Speech Recognizer, which returns an n-best list of possible interpretations. To process sketch input, MIDOS utilizes the Microsoft Ink library to obtain raw stroke data, and then a sketch recognition framework developed by the Multimodal Understanding Group (Sezgin, Stahovich, & Davis, 2001) to recognize the strokes. The stroke recognizer returns a ranked list of possible interpretations for the stroke. Possible interpretations currently supported by the framework are lines, arcs, polylines, ellipses, and complex shapes composed of both lines and arcs.

2.1.2.2 Output synthesis

MIDOS uses the AT&T Natural Voices Speech Synthesizer to generate speech, and produces sketch output by generating strokes with an internal stroke synthesizer that are then displayed through the use of the Microsoft Ink library.

2.1.2.3 User Interface

The MIDOS user interface is shown in Figure 2-4. From the top toolbar, the user can select from several different pen and highlighter colors, while the bottom of the interface displays the speech output from the system and the speech from the user. The interface is written in C# and contains the previously described components that are responsible for acquiring the speech and sketch input and generating the speech and sketch output. MIDOS communicates with the rest of the system, which is written in Java, via a standard socket connection (Adler, 2009).

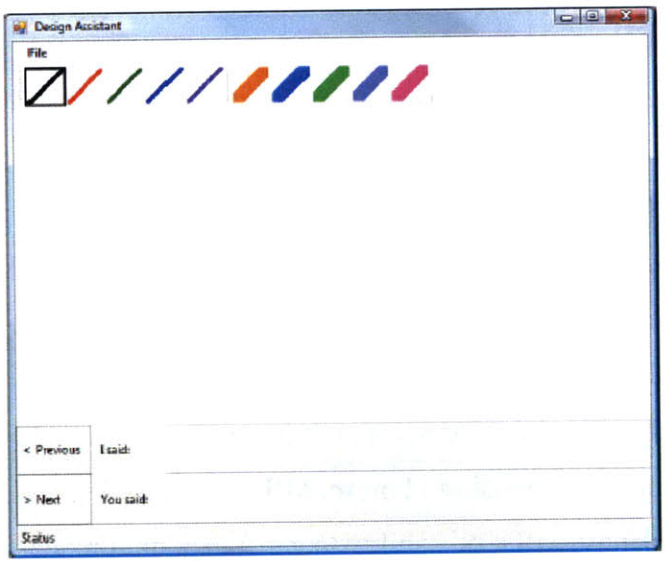


Figure 2-4: The MIDOS user interface (Adler, 2009)

2.1.2.4 Qualitative Physics Simulator

The physics simulator takes the current state of the device and attempts to simulate the rest of the device's behavior. However, if the current state is missing certain pieces of information, the simulator will be unable to complete the simulation. In that event, the system generates an information request for each missing piece of information. These information requests will eventually result in MIDOS asking questions to the user in order to fill in the missing information and complete the simulation (Adler, 2009).

Whenever the physics simulator creates new information requests, the question selection component selects one of them, forms a question from the information request and then sends the question to the output synthesis component (Adler, 2009).

The dialogue control component handles many important management functions in MIDOS. For instance, it is responsible for processing the user's input, and determining whether the user has sufficiently answered the question. If a sufficient answer has been provided, the dialogue control component will update the current state of the physics simulator. If not, it will ask the user a follow-up question (Adler, 2009).

2.2 Limitations of MIDOS

There are many ways in which MIDOS can be improved to become more flexible and support a richer interaction between the user and computer. First, MIDOS is only able to recognize strokes one at a time. This is especially a problem if we want to do handwriting recognition and more complicated shape recognition into MIDOS because a printed single word or shape can be composed of many different strokes. Second, MIDOS lacks simple shape recognition that would allow the user to draw the initial structure of the mechanical device, rather than having to input the information manually. Third, MIDOS does not permit the user to edit the state of the mechanical device during the interaction. Fourth, MIDOS is unable to recognize handwriting. Finally, and most importantly, MIDOS's ability to generate a multimodal dialogue is restricted to the domain of mechanical devices.

2.3 Modifying and Extending MIDOS

One of the main contributions of this thesis is to modify and extend the original MIDOS so as to address the issues raised and discussed in Section 2.2. The remainder of this section describes the specific changes that have been made to MIDOS and how those changes dealt with these issues.

2.3.1 Multiple Stroke Recognition

The original MIDOS was able to handle sketch recognition of only one stroke at a time (the process described in Section 2.1.2.1). We had to modify this process to support the recognition of objects that are composed of multiple strokes. In particular, we have modified the new input acquisition component so that it employs a one second temporal window to group strokes together for recognition. If a new stroke is started within one second after the end of the last stroke, it will be added to a list of grouped strokes. If the system sees more than a one second delay after the last stroke, the input acquisition component sends the entire list of grouped strokes to be classified together.

For historical reasons, the multi-stroke recognizers we added to MIDOS do not take into account that each individual stroke has already been classified as a line, arc, polyline, ellipse, or complex shape composed of both lines and arcs (as described in Section 2.1.2.1). It should be

possible in the future to modify the system so that the recognizers use an approach, similar to Hammond's LADDER system (Hammond & Davis, 2005), that can take advantage of the fact that each stroke has already been classified individually.

2.3.2 Shape Recognition

The original MIDOS framework required the user to input the initial state of the domain model (*i.e.,* the starting state of a mechanical device) manually before the system would be able to carry out a dialogue with the user. In order to allow MIDOS to handle an initial sketch of the domain state model that is composed of simple shapes, we integrated another sketch recognition framework into MIDOS that was developed by the Multimodal Understanding Group (Hammond & Davis, 2002). As a result, MIDOS is now capable of recognizing rectangles, diamonds, open-headed arrows, triangle-headed arrows, and diamond-headed arrows. We also created a separate recognizer that recognizes scribbles by looking for polylines with more than 15 vertices. Figure 2-5 illustrates all the new shapes.

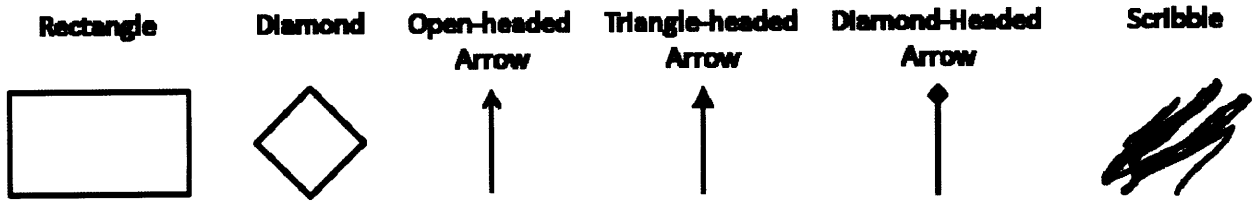


Figure 2-5: The new shapes that have been incorporated into the MIDOS sketch recognizer.

2.3.3 Editing the Domain State

With the addition of the new sketch recognizers described in Section 2.3.1, the user is now able to sketch the domain state with MIDOS, as well as modify the state of the domain during their dialogue with the computer. For example, consider again the spring-block device discussed in Section 2.1. It might be the case that after MIDOS asks the user about the direction of the spring, the user realizes that there should be another block to the right of the original block, as shown in Figure 2-6. By adding a background information request that listens for changes to the state of the domain, the user is now able to edit the domain state.

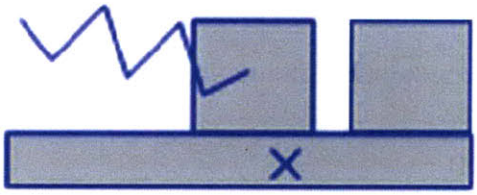


Figure 2-6: The modified spring-block mechanical device.

2.3.4 Adding Handwriting to MIDOS

With the modification to the input acquisition component that allows the grouping and classification of multiple strokes, we were able to incorporate handwriting recognition into MIDOS. Handwriting is considered a new mode of communication in MIDOS and, as such, is recognized separately from speech and sketch. This section outlines how handwriting is now recognized and used in MIDOS.

2.3.4.1 Recognizing Handwriting

First, we utilize the Microsoft Handwriting Recognizer installed with Windows XP Tablet PC Edition to return a ranked n-best list of interpretations of the possible handwriting text. MIDOS then calculates a *HandwritingScore* for each of the possibilities. That score is calculated by comparing the handwriting input with the list of phrases and their locations that the system expects in response to the question it asked. The *HandwritingScore* for each entry is calculated as:

$$HandwritingScore = 100 * RankingScore * MatchPercentage$$

The *HandwritingScore* for each entry is scaled by 100 to ensure that it falls between 0 and 1000, and therefore matches the scoring metric used for speech and sketch. It is also weighted by the entry's ranking on the n-best list with the top entry receiving a *RankingScore* of 10 and each subsequent entry's *RankingScore* decreasing by 1. Finally, an entry's *MatchPercentage* represents the degree to which the entry matches the expected input in accordance with this calculation:

$$MatchPercentage = \begin{cases} ExpectedWords = 0 & : & LocationFactor \\ ExpectedLocations = 0 & : & WordPercentage \\ o.w. & : & 0.5 * WordPercentage + 0.5 * LocationFactor \end{cases}$$

$$LocationFactor = \begin{cases} 1 : \text{ if at least one of the handwriting strokes intersects one of the } ExpectedLocations \\ 0 : \text{ o.w.} \end{cases}$$

$$WordPercentage = \frac{MatchedWords - 0.5 * ExtraWords}{ExpectedWords}$$

In the calculation, *MatchedWords* is the number of words in the entry's recognized text that match the words in the expected handwriting text, *ExtraWords* is the number of words in the entry's recognized text that do not match any word in the expected handwriting text, *ExpectedWords* is the number of words in the expected handwriting text, and the *ExpectedLocations* is composed of the different locations where the system is expecting handwriting input.

As the calculation for *MatchPercentage* illustrates, there are three types of expected handwriting input that the system can handle: (1) specific handwritten phrases that occur only in specific locations, (2) specific handwritten phrases that can occur anywhere, and (3) any handwritten phrase that occurs only in specific locations. We will now proceed to illustrate the manner in which MIDOS is able to recognize new handwriting input according to each of these three types of expected handwriting input.

Recognizing Handwriting Input Example 1: Phrases and Locations

The first example involves the manner in which MIDOS handles handwriting input when it is expecting a specific phrase at a specific location. This type of expected handwriting is utilized in the task in the database normalization domain (described below in Chapter 5) in which the user selects attributes to be the primary key for an entity. In order to identify an attribute as being part of the primary key, the user can write the text "primary key" next to the name of the attribute.

For instance, Figure 2-7 shows the *Name* attribute in red, which is composed of the red ellipse and red text. It also illustrates the input that the system is expecting for the case of selecting *Name* as a primary key, which includes *ExpectedWords* = "primary key" (shown in black) and *ExpectedLocations* = red ellipse. Figure 2-8 illustrates the actual user's input.

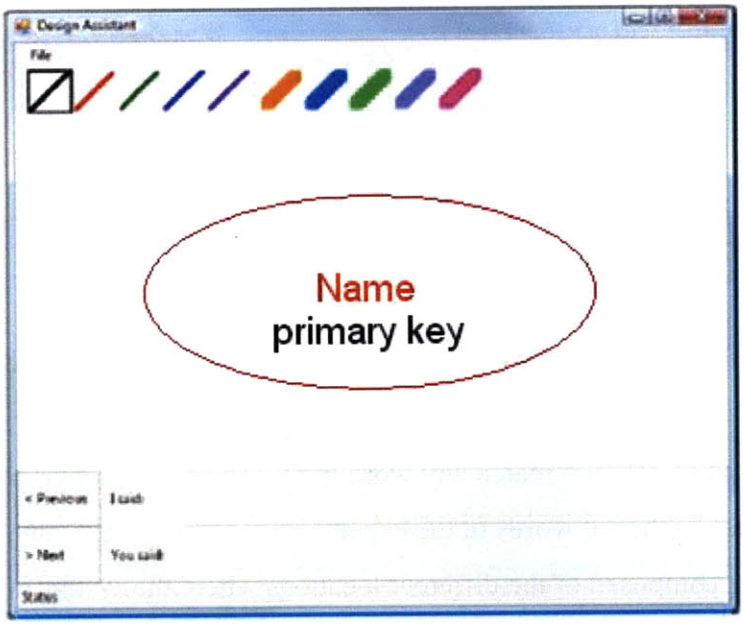


Figure 2-7: An illustration of the first type of expected handwriting that MIDOS can handle.

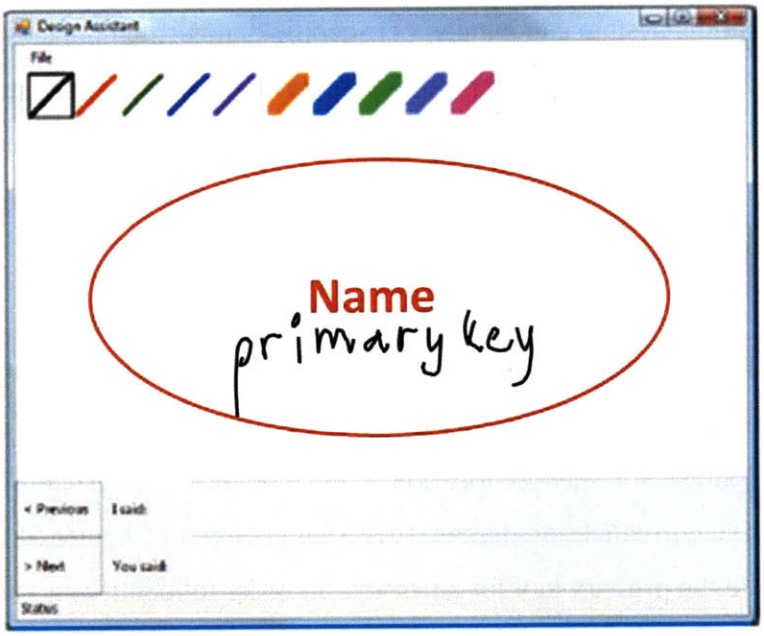


Figure 2-8: The user's input for Example 1.

The system recognizes and scores the user's input according to its expected input by sending the stroke input (shown in black in Figure 2-8) to the Microsoft Handwriting Recognizer, which then returns an n-best list of recognized handwriting. The top 5 results of the n-best list are show in Table 2-1. For each of these results, the system then calculates a

HandwritingScore – in this example, since *ExpectedWords* $\neq 0$ and *ExpectedLocations* $\neq 0$, the *HandwritingScore* is calculated as:

$$HandwritingScore = 100 * RankingScore * (0.5 * WordPercentage + 0.5 * LocationFactor)$$

Table 2-1: Microsoft Handwriting Recognizer n-best list

Rank	Recognized Handwriting
1	primary key
2	primary Very
3	primary kery
4	primary Viey
5	primary Very

Table 2-2 illustrates the *HandwritingScore* for each of the top 5 entries on the n-best list, as well as the intermediate calculations. The first entry on the list receives the best score (a perfect score of 1000) because its recognized handwriting exactly matches the *ExpectedWords* and falls within the one *ExpectedLocation*.

Table 2-2: The scoring calculations for each entry on the n-best list

Ranking	Recognized Handwriting	RankingScore	WordPercentage	LocationFactor	HandwritingScore
1	primary key	10	1	1	1000
2	primary Very	9	0.25	1	562.5
3	primary kery	8	0.25	1	500
4	primary Viey	7	0.25	1	437.5
5	primary Very	6	0.25	1	375

Recognizing Handwriting Input Example 2: Phrases

The second example involves the manner in which MIDOS handles handwriting input when expecting a specific phrase that can occur at any location. This type of expected handwriting can be used to identify handwritten commands from the user. For instance, one way that the user could tell the system to add an attribute is to write "add attribute" anywhere on the screen. In this example the system's expected input would be *ExpectedWords* = "add attribute" and *ExpectedLocations* = 0 (because the handwriting can occur at any location). Figure 2-9 illustrates the user's actual input.

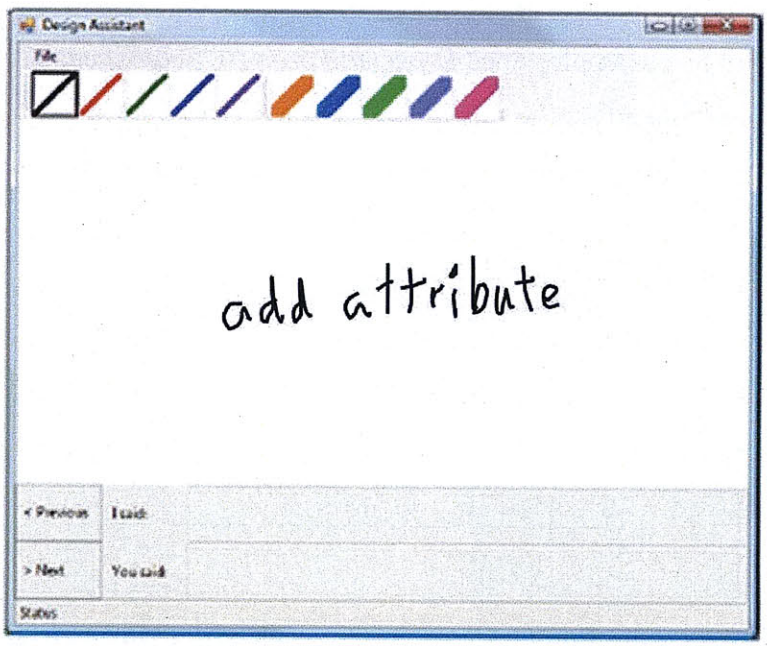


Figure 2-9: The user's input for Example 2.

The top 5 results of the n-best list from the Microsoft Handwriting Recognizer are shown in Table 2-3. For each of these results, the system then calculates a *HandwritingScore* - in this example, since *ExpectedWords* $\neq 0$ and *ExpectedLocations* $= 0$, the *HandwritingScore* is calculated as follows:

$$HandwritingScore = 100 * RankingScore * WordPercentage$$

Table 2-3: Microsoft Handwriting Recognizer n-best list for Example 2

Rank	Recognized Handwriting
1	add attribute
2	add attribate
3	add attrib ate
4	add attrib Ute
5	add attrib cite

Table 2-4 illustrates the *HandwritingScore* for each of the top 5 entries on the n-best list, as well as the intermediate calculations. The first entry in the list receives the best score (a perfect score of 1000) because its recognized phrase matches the *ExpectedWords*.

Table 2-4: The scoring calculations for each entry on the n-best list for Example 2

Ranking	Recognized Handwriting	RankingScore	WordPercentage	HandwritingScore
1	add attribute	10	1	1000
2	add attribate	9	0.25	225
3	add attrib ate	8	0	0
4	add attrib Ute	7	0	0
5	add attrib cite	6	0	0

Recognizing Handwriting Input Example 3: Locations

The third example involves the manner in which MIDOS handles handwriting input when it is expecting any handwriting at a specific location. This type of expected handwriting is utilized in the task in the database normalization domain (that we will describe later in Chapter 5) in which the user needs to input the name for a new, blank entity. In order to supply the system with the entity's name, the user writes the name inside the entity's empty rectangle.

For instance, Figure 2-10 shows an unnamed entity in red and illustrates the expected input if the user were to write a name for this entity, which includes the area outlined by the red rectangle as the *ExpectedLocation* and no *ExpectedWords*. Figure 2-11 illustrates the user's actual input.

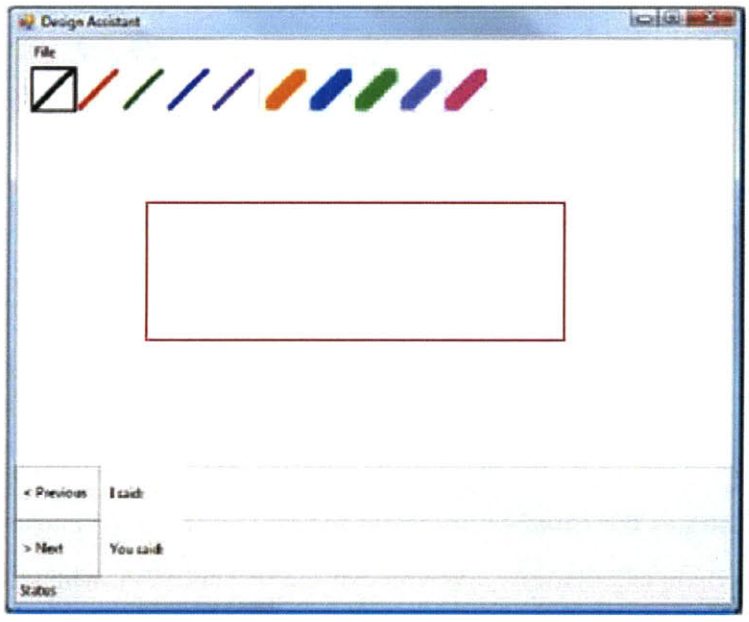


Figure 2-10: An illustration of the third type of expected handwriting that MIDOS can handle.

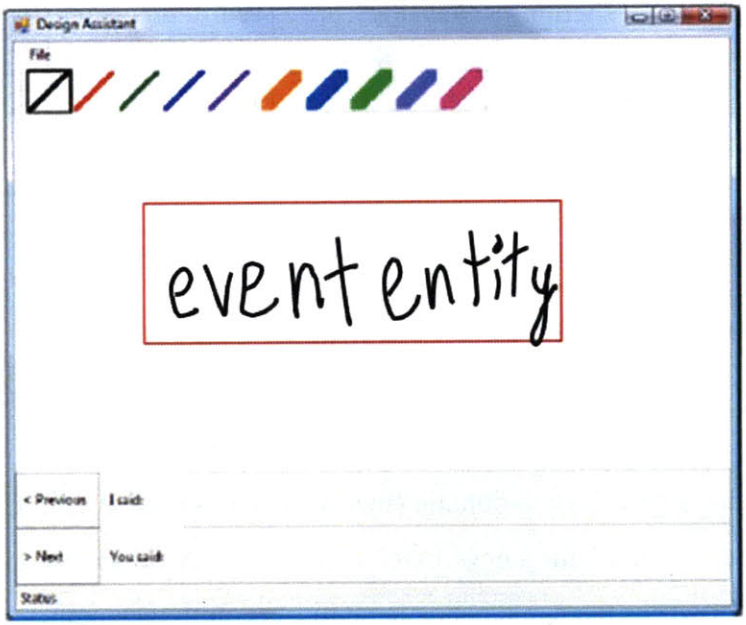


Figure 2-11: The user's input for Example 3.

The top 5 results of the n-best list from the Microsoft Handwriting Recognizer are shown in Table 2-5. For each of these results, the system then calculates a *HandwritingScore* - in this example, since *ExpectedWords* = 0 and *ExpectedLocations* ≠ 0, the *HandwritingScore* is calculated as follows:

$$HandwritingScore = 100 * RankingScore * LocationFactor$$

Table 2-5: Microsoft Handwriting Recognizer n-best list for Example 3

Rank	Recognized Handwriting
1	event entity
2	event Entity
3	event catty
4	event citify
5	event entirety

Table 2-6 illustrates the *HandwritingScore* for each of the top 5 entries on the n-best list, as well as the intermediate calculations. As you can see in Table 2-6, when the system is expecting the type of handwriting input displayed in this example (where *ExpectedWords* = 0 and *ExpectedLocations* ≠ 0), it scores each of the entries according to their rank on the n-best list.

Table 2-6: The scoring calculations for each entry in the n-best list for Example 3

Ranking	Recognized Handwriting	RankingScore	LocationFactor	HandwritingScore
1	event entity	10	1	1000
2	event Entity	9	1	900
3	event catty	8	1	800
4	event citify	7	1	700
5	event entirety	6	1	600

2.3.4.2 Processing Recognized Handwriting

Since handwriting is treated as a new mode of communication in the MIDOS system on the same level as speech and sketching, we needed to modify the method by which MIDOS combines and processes multimodal input. Outlined below is the method that the revised version of MIDOS uses to process a user's multimodal input and to update the state of the domain, as modified to incorporate handwriting input (Adler, 2009):

1. Ask the user a question.
2. Group user's speech, sketching and handwriting.
3. Match and score the user's speech against the expected speech.
4. Match and score the user's sketching against the expected sketching.
5. Match and score the user's handwriting against the expected handwriting.
6. Find the combination of speech, sketching, and handwriting input that maximizes the sum of their respective speech, sketching, and handwriting scores.
7. Evaluate the best scoring combination
    a. If the combination is successful, go to the next step.
    b. If the combination is unsuccessful, ask the user a follow-up question with more guidance about the expected answer. Return to the first step.
8. Update current state based on the new information.

2.4 A Domain-Independent MIDOS

The primary goal of this thesis is to take the capabilities of MIDOS and apply them to other domains. This was a difficult task to accomplish because the domain-specific knowledge (*i.e.*, knowledge about mechanical devices) of MIDOS was too tightly coupled with the management of the dialogue. We therefore needed to modify the architecture of MIDOS so that the system could become domain independent. Figure 2-12 illustrates the new MIDOS architecture.

In the new architecture, all of the components that rely on knowledge about the domain, including the current state of the domain and possible information requests about the domain, have been decoupled from the rest of the system and reside in a new component named the Domain Backend. The Domain Backend interacts with the rest of MIDOS by way of a defined interface called the MIDOS Interface. In the example of the mechanical device domain, the Qualitative Physics Generator and Question Generator components are integrated into the Domain Backend. Consequently, in order to allow MIDOS to generate a multimodal dialogue in another domain, you need to only replace the Domain Backend component, with a new Domain Backend that implements the MIDOS Interface. The next section describes the MIDOS Interface in more detail.

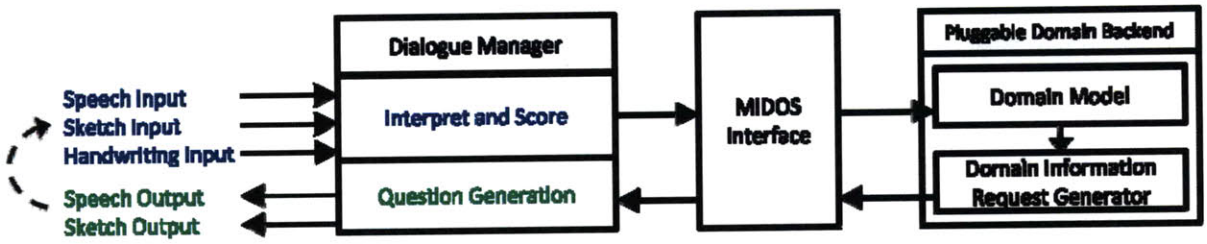


Figure 2-12: An illustration of the new MIDOS architecture.

2.4.1 MIDOS Interface

The MIDOS Interface acts as a bridge between the Dialogue Manager and the domain-specific knowledge stored in the Domain Backend. The MIDOS Interface uses objects called information requests to encapsulate all of the information concerning an interaction between the system and user with respect to a particular piece of domain knowledge. This means that an information request object stores the multimodal question that the system will ask the user regarding the domain knowledge, as well as the multimodal responses that the system can expect from the user. It also knows how to update the domain model when it receives an acceptable response from the user. These information request objects play an integral role in the manner in which MIDOS controls the interaction between the computer and the user.

Figure 2-13 illustrates how the new MIDOS system processes a user's multimodal input and the role that the MIDOS Interface plays in this process. We describe the functionality of the MIDOS Interface by describing each of the 8 steps involved in processing the user's input and generating the system's next question. .

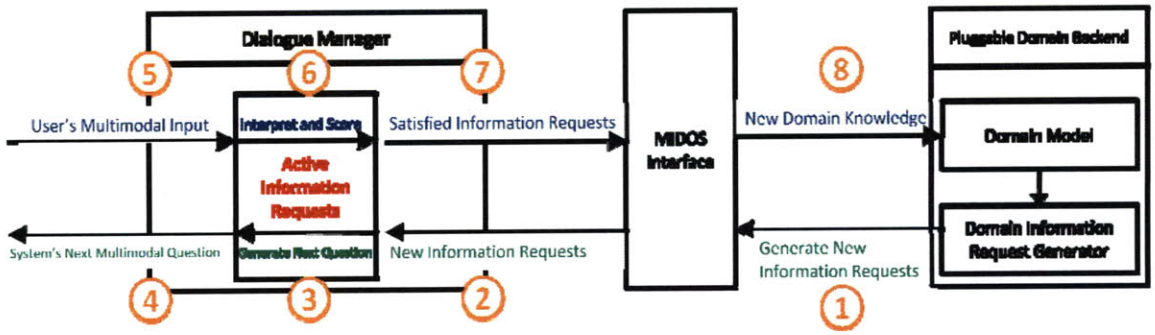


Figure 2-13: A step by step illustration of the flow of information in MIDOS through the new MIDOS Interface.

1) The MIDOS Interface gets a new set of information requests from the Domain Backend, each of which corresponds to a piece of information that the user needs to supply to the system in order for the domain model to be complete.
2) The MIDOS Interface passes these new information requests to the Dialogue Manager.
3) The Dialogue Manager generates a new question from one of the previously active information requests or one of the new information requests that it just received from the MIDOS Interface.

4) MIDOS passes the new question to output synthesizers that produce the speech output and sketch output for the user.
5) The Input Acquisition component captures the user's multimodal response and passes it to the Dialogue Manager.
6) The Dialogue Manager attempts to determine if the user has provided a sufficient answer to one of the system's previous questions. This is accomplished by matching the user's multimodal input against the expected input of each active information request. An information request is activated when the system asks the information request's question.
7) The Dialogue Manager takes the active information requests whose expected input matched the user's input and sends them to the MIDOS Interface.
8) The MIDOS Interface takes each satisfied information request and uses it to update the domain model with new domain knowledge.

The MIDOS Interface is composed of two classes, the *InformationRequest* class and the *BackendInterface* class. For every type of question that the system can ask the user, the developer must create a separate *InformationRequest* class. This class must extend the *InformationRequest* abstract class and overwrite the *generateQuestion, getExpectedHandwriting*, *getExpectedStrokes*, *getExpectedSpeech*, and *setFacts* methods. The *generateQuestion* method is called by the Dialogue Manager component when it activates the *InformationRequest* and needs to ask the user a question about the *InformationRequest*. The *getExpectedHandwriting*, *getExpectedStrokes*, and *getExpectedSpeech* methods are called by the Dialogue Manager in order to determine if it user's actual input matches the *InformationRequest's* expected multimodal input. If it does match, then the Dialogue Manager calls the *InformationRequest's* setFacts method, which updates the state of the domain with the information.

For instance, we created an *UnnamedEntityInformationRequest* for our database design system to allow the system to ask the user about unnamed entities in the ER model. An abbreviated pseudo-code version of this class has been included below, which displays the behavior of the overridden methods:

```
UnnamedEntityInformationRequest extends InformationRequest {
      …
      @Override
      public Question generateQuestion(…) {
            1. Set the question's speech to:
               "Please write in a name for this entity"
```

```
            2. Set the question's strokes to include a circling stroke around
               the entity
        }

        @Override
        public List<ExpectedHandwriting> getExpectedHandwriting() {
              Add the second type of Expected Handwriting described in Chapter
              2.3.4.1, where the ExpectedLocations includes the rectangle of
              the unnamed entity and there are no ExpectedWords.
        }

        @Override
        public List<ExpectedStrokes> getExpectedStrokes() {
              return new ArrayList<ExpectedStrokes>();
        }

        @Override
        public List<ExpectedSpeech> getExpectedSpeech() {
              return new ArrayList<ExpectedSpeech>();
        }

        @Override
        public void setFacts(…) {
              Rename the entity in the entity-relationship model according to
              the user's input
        }
        …
}
```

In addition to implementing the individual *InformationRequest* classes, the developer must also create one backend class for the domain of the system that implements the *BackendInterface* interface. This backend must implement the *getActiveInformationRequests* method, which is responsible for returning all the active *InformationRequest* classes according to the current state of the domain.

For instance, we created a *DatabaseDesignBackend* class for our database design system that stores all the domain-specific information, including the ER model, and implements the *BackendInterface*. Although it is not illustrated below, the *DatabaseDesignBackend* is also responsible for displaying the current state of the domain model via the User Interface component. For this class, the *getActiveInformationRequests* method is responsible for looking through the ER model, and creating an *InformationRequest* class for every piece of incomplete information in the model. As one example, if the method finds an unnamed entity in the model, it will generate and return a new *UnnamedEntityInformationRequets* class corresponding to that unnamed entity as shown below:

```
DatabaseDesignBackend implements BackendInterface
{
	…
	@Override
	public List<InformationRequest> getActiveInformationRequests(…) {
		…
		for each entity in the entity-relationship model
		{
			if the entity doesn't have a name
			{
				create a new UnnamedEntityInformationRequest
				about that entity
			}
		}
		…
	}
	…
}
```

Chapter 3

Database Design

After creating a more flexible, domain-independent MIDOS (as explained in Section 2.3), our next step was to demonstrate the system's enhanced capabilities by applying MIDOS to the new domain of database design.

This chapter discusses the database design process that we employed, which involved developing an Entity-Relationship (ER) model by creating an ER diagram, then translating the ER model into a Relational Model, which can, in turn, be used to implement a relational database.

3.1 Entity-Relationship Model

In order to start the database design process, it is necessary to build a model of the information that will be stored in the database. One way to do this is to come up with an ER model. An ER model is an abstract representation of data and is composed of objects called entities, relationships, and attributes.

An entity can be thought of as an "object in the real world that is distinguishable from another object" (Ramakrishnan & Gehrke, 2003). An entity can be a physical object (*i.e., electronics store sales clerk*), an event (*i.e. TV sale*), or a concept (*i.e., credit card transaction*). In ER models, it is common to group together similar entities into a single entity-set (*i.e., all electronics store employees*), but for the purposes of this thesis we consider the concept of an entity-set as being synonymous with an entity.

A relationship describes an association between two entities. For example, if there are two entities, an *electronics store sales clerk* entity and electronics *store* entity, we could then define a *works in* relationship that links the two. Each relationship also has a cardinality constraint associated with it that defines the number of entities on one side of the relationship that can be associated with a number of entities on the other side. There are three types of relationship cardinalities: (1) many-to-many, (2) one-to-many, and (3) one-one (Ramakrishnan & Gehrke, 2003).

The differences between each type of cardinality can be illustrated by example. For instance, there is a many-to-many cardinality in the relationship between a *sales clerk* and a *customer* because a *sales clerk* will help many *customers*, and a *customer* may receive assistance from more than one *sales clerk.*

By contrast, there is a one-to-many cardinality in the relationship between a *TV sale* and a *sales clerk* because a *sales clerk* can make many different *TV sales*, but each *TV sale* has just one *sales clerk.*

Finally, there is a one-to-one cardinality in the relationship between a *sales clerk* and a *shift* because each *sales clerk* works one *shift* and each *shift* is worked by only one sales clerk (in this example a shift implies a specific date and time).

It is also possible to impose an optionality constraint onto either side of a cardinality. If the optionality constraint is applied to the "one" side of a cardinality, then it is possible for there to be no instances or only a single instance of the corresponding entity. If the optionality constraint is applied to the "many" side of a cardinality, it is possible for there to be anywhere from zero to many instances of the corresponding entity. For instance, there should be an optionality constraint on the "many" side of the relationship between *sales clerk* and *TV sale* because a *sales clerk* is not guaranteed to make even one *TV sale.*

In addition, in ER models it is common to group together similar relationships into a single relationship-set. However, for the purposes of this thesis, we consider the concept of a relationship-set as being synonymous with a relationship.

Both entities and relationships can have attributes. Attributes represent certain properties about the particular entity or relationship with which that they are associated. For instance, a *sales clerk* entity might have the e*mployee ID number* attribute, and the *works in* relationship might have the *since* attribute.

Each entity is also assigned a primary key. A primary key is the minimal set of attributes whose values uniquely identify a particular entity in the set (Ramakrishnan & Gehrke, 2003). If an entity has an attribute that belongs to the primary key of another entity, then that attribute is considered to be a foreign key.

3.1.1 Diagramming an Entity-Relationship Model

An ER model is often generated by drawing a visual representation of the model called an ER diagram. In this diagram, entities are drawn as rectangles, relationships are drawn as diamonds, attributes are drawn as ovals, and the name of each is displayed in the center of their respective shapes depicted in Figure 3-1.

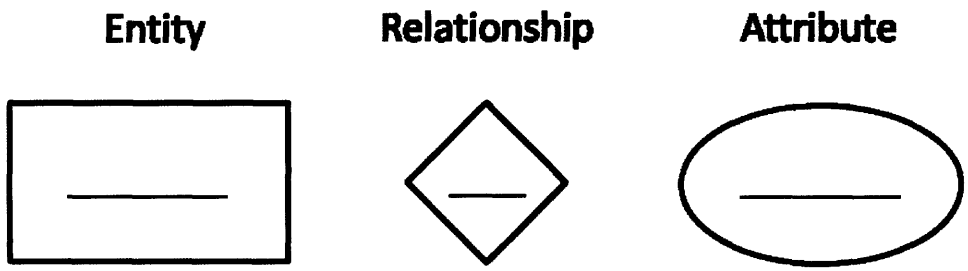


Figure 3-1: The visual representation for entities, relationships, and attributes.

Each attribute is connected to its associated entity or relationship with a line, and if an attribute is part of a primary key, it will have the text "(PK)" inside its oval. Similarly, if an attribute is part of a foreign key, it will also have the text "(FK)" inside its oval. Figure 3-2 shows the visualization of the *electronics store sales clerk* entity that has three attributes: (1) *employee ID number*, (2) *name*, and (3) *electronics store name*. The primary key for the *electronics store sales clerk* entity is the *employee ID number* attribute, and the *electronics store name* attribute is a foreign key attribute for another entity.

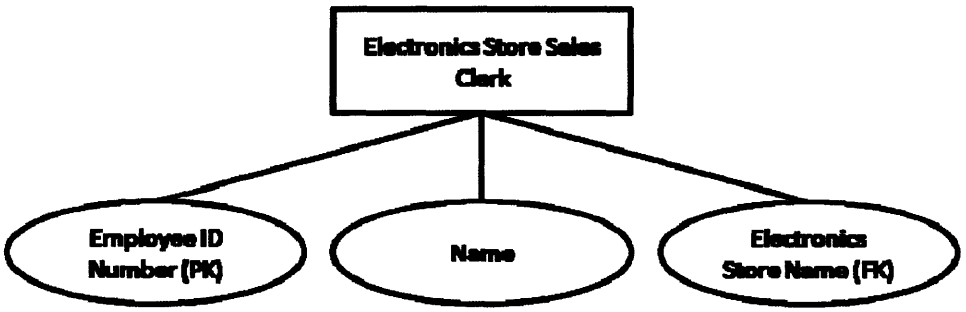


Figure 3-2: A visualization of the *electronics store sales clerk* entity and its attributes.

If an entity is part of a relationship, it is connected to the relationship by a line, with the end of the line shaped according to the cardinality constraint attributed to the entity by the relationship. In this thesis we use Crow's Foot notation (invented by Gordon Everest) to visually represent the cardinality constraint of a relationship. Figure 3-3 shows the manner in which the one-to-many, many-to-many, and one-to-one cardinality constraints are represented using the Crow's foot notation. If there is an optionality constraint attributed to the "one" or "many" side

of a cardinality, the notation is modified as illustrated in Figure 3-4. Additionally, if the side of the a cardinality is unknown, the notation is modified as illustrated in Figure 3-5.

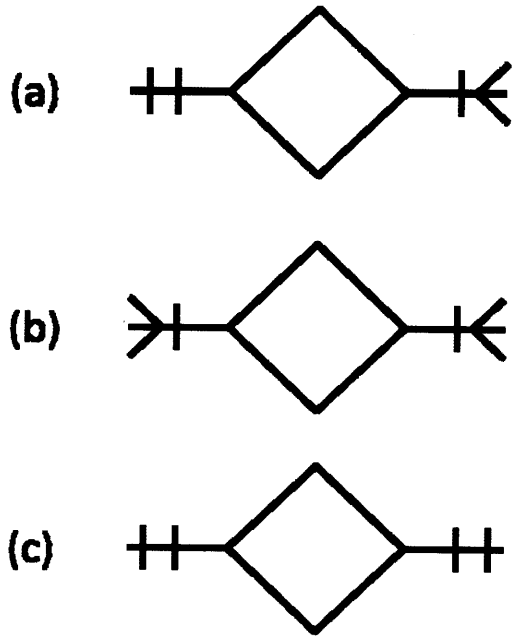


Figure 3-3: A visualization of the three different relationship cardinalities: (a) one-to-many, (b) many-to-many, and (c) one-to-one.

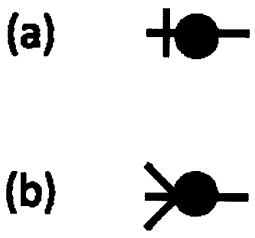


Figure 3-4: A visualization of the optionality constraint on the "one" side of a cardinality (a) and on the "many" side (b).

Figure 3-5: A visualization of an unknown side of a cardinality.

3.1.1.1 Sample ER Diagram

Figure 3-6 illustrates the ER diagram for the ER model that has the following entities and relationships:

Entities

1. *Electronics Store* has the *Address* attribute (primary key).
2. *Sales Clerk* has the *Employee ID Number* attribute (primary key).
3. *TV Sale* has the *TV Product Number* attribute (primary key) and the *Amount* attribute.
4. *Customer* has the *Name* attribute (primary key).

5. *Shift* has the *Start Time* attribute and the *End Time* attribute, which both form its primary key.

Relationships

1. One-to-many *Works in* relationship between *Electronics Store* and *Electronics Store Sales Clerk* that has the *Since* attribute.
2. One-to-many (optionality constraint on many) *Makes* relationship between *Sales Clerk* and *TV Sale* with an optionality constraint on the side of the *Sales Clerk.*
3. Many-to-many (optionality constraint on the second many) *Helps* relationship between *Sales Clerk* and *Customer* with an optionality constraint on the side of the *Customer.*
4. One-to-one *Works* relationship between *Sales Clerk* and *Shift.*

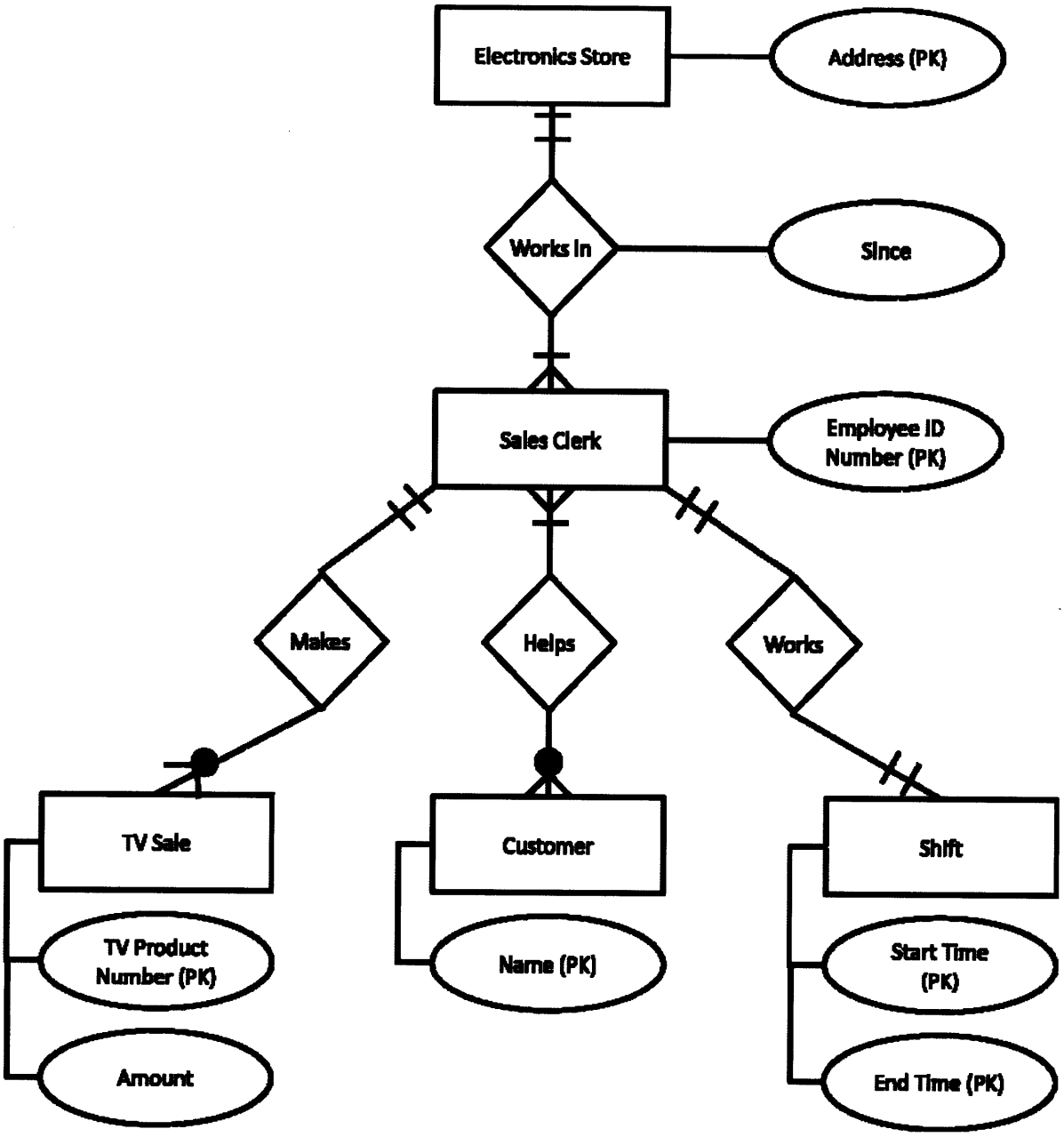


Figure 3-6: An example of an ER diagram.

3.2 Relational Model

The relational model was developed by E.F. Codd in 1969 and forms the foundation for a relational database system. In the relational model, data is represented by relations. "A relation consists of a relation schema and a relation instance" (Ramakrishnan & Gehrke, 2003). The relation instance is a table and the relation schema describes the column heads for the table. Specifically, the relation schema defines the name of the relation and the name and domain for

each column of the relation instance. The domain describes the type of information that is stored in the column (*e.g.*, string, integer, real number). In essence, a relation can be thought of as a table with columns, in which the data stored as rows of the table (Ramakrishnan & Gehrke, 2003).

In addition, each relation has a unique primary key consisting of a column or set of columns that can uniquely identify each row in the table. This means that no two rows can have the same set of values in the primary key column(s). If another relation has a column(s) that refers to the primary key of another relation, then it is called a foreign key, and for every row in that table, the values of the foreign key columns must occur in a single row in the referenced table.

Next, we examine a sample relation that stores all employee information pertaining to an electronics store. For each employee, the relation must store the employee's id number, name, and age in a relational database. The employee information can be represented by the following relation schema:

Employees (*id number*: integer, *name*: string, *age*: integer)

Now that the relation schema has been defined, a sample relation instance can be examined. An instance of the Employees relation can be seen in Table 3-1. A relation instance is defined by a set of data stored as rows in the table.

Table 3-1: An instance of the Employees relation

Columns

Rows

id number	name	age
1000	John	19
1060	Jane	18
1088	Mary	18
1142	Jack	21

3.3 Translation from the Entity-Relationship Data Model to the Relational Model

This section describes how to translate an ER data model into a collection of tables that comprises the relational model.

First, for each entity in the ER model, add a table to the relation. Next, add a column to the table for each of the entity's attributes. The column(s) in the table that correspond to the primary key attributes are defined as the primary key for the table. Second, for each relationship, we take the following steps with its cardinality:

Cardinality	**Translation Rule**
one-to-one	The primary key from the first table is added as a foreign key to the other table, and if the relationship has any attributes, they are added as columns to the first table.
one-to-many	The primary key from the table at the "one" side is added as a foreign key in the table at the "many" side, and if the relationship has any attributes, they are added as attributes to the table on the "many " side.
many-to-many	Add a new intermediate table, where the primary keys from each of the participating entities are added as foreign keys in the new table. The primary key of the intermediate table is the set of all of the columns. And if the relationship has any attributes, they are added as attributes to the intermediate table.
optionality constraint	The values in the rows of the foreign key columns are required.

Figure 3-7 displays the relational model corresponding to the ER data model that is displayed in Figure 3-6.

Electronics Store Table

Address (PK)

Sales Clerk Table

Employee ID Number (PK)	*Address* (FK)	*Since*

TV Sale Table

TV Product Number (PK)	*Amount*	*Employee ID Number* (Required FK)

Sales_Clerk_Customer Intermediate Table

Employee ID Number(PK) (FK)	*Name (PK)* (Required FK)

Customer Table

Name (PK)

Shift Table

Start Time (PK)	*End Time* (PK)	*Employee ID Number* (FK)

Figure 3-7: The corresponding relational model for the ER model displayed in Figure 3-6.

Chapter 4

Database Normalization

Chapter 3 describes how an ER model is used to generate an initial set of relational schemas that represent the conceptual structure of the information to be stored in the database. However, there can be problems with these initial schemas if they call for redundant storage of information. To eliminate redundancy problems, relations are split into smaller relations in a process called decomposition. The decomposition process continues until all the relations meet the criteria of a particular normal form. Constraints called functional dependencies are used to initially identify redundancies and to later drive the decomposition process. This method of refining schema into a normal form is called normalization.

This chapter discusses the different types of redundancy problems that can occur in a relation database and the decomposition process used to fix them. We also introduce functional dependencies and show how it is possible to extract likely function dependencies from the instance of a relation. Finally, we introduce the Boyce-Codd Normal Form (BCNF) and describe how to normalize a relation's schema into BCNF by utilizing functional dependencies.

The contents of this chapter are borrowed largely from Chapter 19 of Ramakirshnan's and Gehrke's book, *Database Management Systems*.

4.1 Redundancy Problems

When the same information is stored repeatedly in a relational database, there are several problems that can arise:

- Update Anomalies: If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated (Ramakrishnan & Gehrke, 2003).
- Insertion Anomalies: It may not be possible to store certain information unless some other, unrelated information is stored as well (Ramakrishnan & Gehrke, 2003).
- Deletion Anomalies: It may not be possible delete certain information without losing some other, unrelated information as well (Ramakrishnan & Gehrke, 2003).

In order to visualize each of these redundancy problems, we will examine one of the examples used in *Database Management Systems*. In particular, consider the following relation's schema:

Hourly_Employees (*ssn*, *name*, *lot*, *rating*, *hourly_wages*, *hours_worked*)

The *ssn* column is the primary key for the *Hourly_Employees* relation, and the *rating* column is directly related to the *hourly_wages* attribute (an example of a functional dependency). This dependency enforces a constraint on the rows of *Hourly_Employees* such that there can be only one unique value of *hourly_wages* for each unique value of *rating*. Table 4-1 depicts an instance of the *Hourly_Employees* relation:

Table 4-1: An instance of the *Hourly_Employees* relation

ssn (PK)	*name*	*lot*	*rating*	*hourly_wages*	*hours_worked*
123-22-366	Attishoo	48	8	10	40
231-21-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Table 4-1 also depicts how the constraint between the *rating* column and *hourly wages* column forces the database to store redundant information; the *rating* value of 8 and *hourly_wages* value of 10 is repeated three times, and the *rating* value of 5 and *hourly_wages* value of 7 is repeated twice. This redundancy also introduces additional redundancy problems:

- Update Anomalies: The *hourly_wages* value in the first row cannot be updated without making a similar change to the same value in the second row (Ramakrishnan & Gehrke, 2003).
- Insertion Anomalies: A new row for a new employee cannot be inserted unless the *hourly*_wages value for the employee's rating value is known (Ramakrishnan & Gehrke, 2003).
- Deletion Anomalies: If all the rows with a given *rating* value (*e.g.,* the rows for Attishoo, Smiley, and Madayan) are deleted, the association between the *rating* value of 8 and the *hourly_wages* value of 10 is lost (Ramakrishnan & Gehrke, 2003).

4.2 Addressing Redundancy Through Decomposition

As illustrated in the *Hourly_Employees* example, simple constraints that exist between the columns of a relation's schema can introduce many redundancy problems. In order to address these problems, a relation of this type can be replaced by several smaller relations in a process called decomposition. "A decomposition of a relation schema R consists of replacing the relation by two (or more) relation schemas that each contain a subset of the attributes of R and together include all attributes in R" (Ramakrishnan & Gehrke, 2003).

For instance, the *Hourly_Employees* relation can be decomposed into:

New_Hourly_Employees (*ssn*, *name*, *lot*, *hours_worked*)

Wages (*rating*, *hourly_wages*)

Table 4-2 and Table 4-3 illustrate the instances of the *New_Hourly_Employees* and *Wages* relations that correspond to the instance of the *Hourly_Employees* relation in Table 4-1.

Table 4-2: An instance of the *New_Hourly_Employees* relation

ssn (PK)	*name*	*lot*	*rating*	*hours_worked*
123-22-366	Attishoo	48	8	40
231-21-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

Table 4-3: An instance of the *Wages* relation

rating(PK)	*hourly_wages*
8	10
5	7

By decomposing the *Hourly_Employees* relation, we have eliminated the redundancy problems noted above.

4.3 Functional Dependencies

Functional dependencies (FD) are constraints that capture dependencies between sets of columns in a relation. *Database Management Systems* defines FDs in the following manner:

Let R be a relation schema and let X and Y be nonempty sets of [columns] in R. We say that an instance z of R satisfies the FD X → Y if the following holds for every pair of [rows] r_1 and r_2 in z:

If $r_1.X = r_2.X$, then $r_1.Y = r_2.Y$

($r_1.X$ represents the values of the columns in X for row r_1)

Table 4-4 shows a relation instance that has the FD AB→ C. The first two rows of the table show that when rows have the same values in columns A and B, they have the same values for column C. The third and fourth rows show that if rows have different values in columns A or B, then they can have a different value in column C. If we were to add the row (*a1,b1,c2,d1*), then the instance would violate the FD because the values *a1* and *b1* in columns A and B, respectively, would not guarantee a value of *c1* in column C (Ramakrishnan & Gehrke, 2003).

Table 4-4: An instance that satisfies the FD: AB → C (Ramakrishnan & Gehrke, 2003)

A	*B*	*C*	*D*
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

4.3.1 Extracting Functional Dependency Candidates from a Relation Instance

As describes below in Chapter 5, there is a need to be able to extract possible candidates for FDs for a relation. It is possible to do so by way of the following algorithm:

1. For each possible subset c of the set of columns
2. And for each row r,
3. Find all the other rows r_{match} that have same values for the columns in c.
4. If there is a set of columns not in c, called c^*, for which r and r_{match} have the same values, then add a possible candidate FD from $c \rightarrow c^*$.

For example, look at the behavior of this algorithm on the relation instance shown in Table 4-4. The set of possible values for c for this relation are: {A}, {B}, {C}, {D}, {AB}, {AC}, {AD}, {BC}, {BD}, {CD}, {ABC}, {ABD}, {ACD}, and {BCD}.

First, consider c = A. There are two possible values of A in the relation instance, *a1* and *a2*, but the algorithm disregards *a2* because it occurs only in one row. Since the value *a1* occurs in rows 1-3, the algorithm checks to see if there is any combination of other columns that has the same values for rows 1-3, but does not find any such combination. The algorithm therefore determines that there are no FDs stemming from A.

Next, consider c = AB. There are three possible values of AB in the relation instance, *a1 b1*, *a2 b1*, and *a1 b2*, but the algorithm disregards *a2 b1* and *a1 b2* because they each only occur in one row. The value *a1 b1* occurs in rows 1 and 2, so the algorithm checks if there is any other combination of other columns that has the same value for rows 1 and 2, and finds a match with combination {C}. Therefore, the algorithm adds the following FD candidate: AB →C (which happens to be the FD illustrated by the sample instance).

Overall, the algorithm finds 4 possible FD candidates out of the relation instance (see Table 4-5), including AB→C, which is the FD illustrated by the sample instance. In this case, the set of the candidate FDs, C→A and C→B, happens to be equivalent to AB→C due to the reflexivity property of FDs.

Table 4-5: The possible FDs that can be extracted from the relation instance in Table 4-4

	c	*c**
1	C	A
2	C	B
3	AB	C
4	BC	A

4.4 Boyce-Codd Normal Form

It can be difficult to determine if it is necessary to decompose a relation into smaller relations. To address this problem, we use a normal form. If a relation's schema meets the criteria of a specific normal form, then the relation can be known to have the properties defined by that

normal form. This section describes Boyce-Codd normal form (BCNF), and the normalization process.

Database Management Systems defines BCNF as:

> Let R be a relation schema, F be on the set of FDs given to hold over R, X be a subset of [columns] of R, and A be [a column] of R. R is in Boyce-Codd normal form if, for every FD X → A in F, one of the following statements is true:
>
> - $A \in X$
> - X is the primary key

If a relation schema does not meet the conditions of BCNF, then it is possible to decompose the schema into smaller BCNF schemas in a process called normalization. We use the following normalization algorithm described in *Database Management Systems*:

1. For a relation schema *R* with the FDs *F*,
2. Suppose that *R* is not in BCNF. Let $X \subset R$, *A* be a single [column] in *R*, and $X \rightarrow A$ be a FD that causes a violation of BCNF. Decompose *R* into *R-A* and *XA*.
3. If either *R-A* or *XA* is not in BCNF, decompose them further by a recursive application of this algorithm.

R-A denotes the set of columns other than *A* in *R*, and *XA* denotes the union of columns in and *A*.

Next, consider a relation *R* that has columns *A, B,* and *C*, where column *A* also represents the primary key for the relation. In addition, *R* has the FD $C \rightarrow B$. Since *C* is not a part of the primary key (*A*), this FD causes a violation of BCNF. According to the algorithm, if we want to normalize *R* we need to decompose it into one relation that has columns *A* and *B* (*R-A*), and another than has columns *C* and *A* (*CA*). Both of these new relations are in BCNF. Table 4-6 shows an instance of *R* and Table 4-7 and Table 4-8 show that data is stored in *R-A* and *CA*, respectively.

Table 4-6: An instance of relation *R*

A	*B*	*C*
a1	b1	c1
a1	b1	c2
a2	b3	c3
a3	b2	c4
a2	b3	c3

Table 4-7: An instance of the relation *R-A* which was created from the decomposition of relation *R*

A	*B*
a1	b1
a2	b3
a3	b2

Table 4-8: An instance of the relation *CA* that was created from the decomposition of relation *R*

C	*A*
c1	a1
c2	a1
c3	a2
c4	a3

By decomposing *R* into *R-A* and *CA*, both of which are in BCNF, we have eliminated the possibility for update anomalies, insertion anomalies, and deletion anomalies.

Chapter 5

After-Action Report Wiki Domain

An after-action report is used by the military as a way for soldiers to report on the events they experienced while on patrol. These events can be wide-ranging, *e.g.,* from interviews with the local population, to the manner in which to spot possible ambush locations, to the details of a skirmish. The after-action reports provide valuable intelligence for both the patrol commanders, and the individual soldiers who will be participating in similar, future patrols. The most common method of creating such a report is by preparing a large paper document. This inefficient method is inconsistent with the intended purpose of the after-action report because the information contained in the report is not readily accessible by the commanders or other soldiers, In addition, the reports require a great deal of time and effort to produce, which results in the preparation of fewer reports and a related loss in intelligence.

Recently, the military has discovered a simple replacement for the paper after-action reports: wikis. Different military bases have set up their own after-action report wikis that soldiers use to store information about their daily patrols and duties. The wiki offers several advantages over paper reports. First, information can be searched at greater speed and ease. Second, the location of specific events can be graphically displayed on a map, making it easier to assimilate the information. Finally, the wiki provides a much faster and easier way for soldiers to input information, which results in more soldiers submitting larger amounts of data.

Unfortunately, there is no central wiki server that can be accessed by these individual bases due to the remoteness of many of their locations. As a result, each military base must implement its own local wiki, which can be difficult for soldiers with limited programming experience. However, creating a wiki can be much easier if the soldiers are assisted by a computer program design assistant. It is for that reason that we chose to apply MIDOS to the problem of creating an after-action report wiki. In particular, we will demonstrate how MIDOS assists the user in generating the backend database for storing events for a wiki.

5.1 MIDOS Relational Database Design Process

The system that we created utilizes the MIDOS framework to assist a user in generating a database design, and then utilizes automated scripts to generate the database. First, MIDOS helps the user to build an entity-relationship model to represent data that they needs to be stored in the database by sketching an entity-relationship diagram (see Chapter 3.1). As part of this process, MIDOS points out to the user any flaws or mistakes in the model. When the user and system are satisfied with the design of the entity-relationship model, MIDOS then translates it into a relational model (see Chapter 3.3). Next, the system utilizes a script developed by BAE Systems to implement and run a relational database using the relational model. At this point, then, the relational database is ready to store information (*e.g.,* for an after-action report wiki).

After the database has been used to store data, the system has additional features that allow the user to reformat the database in order to eliminate storing redundant data. The system uses the algorithm described in Chapter 4.3.1 to extract possible FD candidates from the instances of the relation model, and communicates with the user to determine if any of the candidates represent actual FDs. If any of the new FDs violates BCNF, then the system will perform normalization on the database to eliminate the problem. The more data stored in the relation database, the better the system will be able to predict previously unknown FDs.

5.2 What does MIDOS have to know about Database Design?

In order to assist the user with the generation of the entity-relationship model and the eventual normalization of that model, we needed to supply MIDOS with information about the database design domain. Having modified MIDOS to be relatively domain-independent (see Chapter 2.4), the only additions we had to make were to create a *DatabaseDesignBackend* that implements the *BackedInterface* and various *InformationRequest* classes corresponding to all the different types of questions that we want the system to be capable of asking the user.

5.2.1 DatabaseDesignBackend

The *DatabaseDesignBackend* class is responsible for storing the current state of the entity-relationship model that the user is building, and displaying the model on the User Interface as an entity-relationship diagram. Since the *DatabaseDesignBackend* implements the *BackendInterface* interface (described in Chapter 2.4.1), it also contains the method *getActiveInformationRequests*. This method looks at the current entity-relationship model and generates the appropriate *InformationRequests*. The specific types of *InformationRequests* utilized in the database design domain are described below.

5.2.2 Database Design Domain InformationRequests

This section describes each class of *InformationRequest* used in the database design domain, including those conditions in the entity-relationship model that activate the request, the system questions stored by the request, and the expected speech, sketch, and handwriting for the user's response.

ListenerInformationRequest

In the current scenario, MIDOS allows the user to build an ER diagram without any interruptions from the system. However, the system does supply a notification to the user (using the *ListenerInformationRequest*) when it identifies a problem with the current design. Whenever the *DatabaseDesignBackend* finds any of the following conditions in the current ER model, it activates the *ListenerInformationRequest*:

- An unnamed entity, relation, or attribute.
- An attribute that is not associated with an entity.
- An entity that has not been assigned a primary key.
- A relation that is not associated with two entities.
- A relation that has not been assigned a cardinality.

The *ListenerInformationRequest* places a red question mark in the upper left corner of the User Interface to notify the user that something is wrong with the current design, and then waits for the user to ask the system for help. The specific help speech that is expected is shown in Table 5-1. The *ListenerInformationRequest* does not have any expected speech or handwriting input.

Table 5-1: *LlstenerInformationRequest* expected help speech

Help Speech
go ahead
what's the problem
what's wrong

When the *ListenerInformationRequest* identifies that the user has asked for help, it notifies the rest of the system that it is okay to ask the user questions. MIDOS will then use *InformationRequests* to query the user, until all remaining problems are eliminated from the ER model.

UnnamedEntityInformationRequest

The *UnnamedEntityInformationRequest* is activated when the ER model contains an entity that has not been named. The multimodal question associated with this request is shown in Figure 5-1. The system expects the user to respond with handwriting inside the identified entity's displayed rectangle. An example of an acceptable user response is shown in Figure 5-2. After the *UnnamedInformationRequest* receives an acceptable response from the user, it will rename the entity in the ER model to the recognized handwriting's text.

Figure 5-1: Screenshot of the multimodal question associated with an *UnnamedEntityInformationRequest* where the system asks the user "What is the name of this entity?" while identifying the unnamed entity with a blue circle.

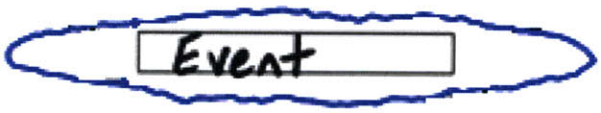


Figure 5-2: An example of acceptable user input for the *UnnamedEntityInformationRequest*.

UnnamedAttributeInformationRequest

The *UnnamedAttributeInformationRequest* is activated when the ER model contains an attribute that has not been named. The multimodal question associated with this request is shown in Figure 5-3. The system expects the user to respond with handwriting inside the identified

attribute's displayed ellipse. An example of an acceptable user response is shown in Figure 5-4. After the *UnnamedAttributeInformationRequest* receives an acceptable response from the user, it will rename the attribute in the ER model to the recognized handwriting's text.

Figure 5-3: Screenshot of the multimodal question associated with an *UnnamedAttributeInformationRequest* where the system asks the user "What is the name of this attribute?" while identifying the unnamed attribute with a blue circle.

Figure 5-4: An example of acceptable user input for the *UnnamedAttributeInformationRequest*.

UnnamedRelationInformationRequest

The *UnnamedRelationInformationRequest* is activated when the ER model contains a relation that has not been named. The multimodal question associated with this request is shown in Figure 5-5. The system expects the user to respond with handwriting inside the identified attribute's displayed ellipse. An example of an acceptable user response is shown in Figure 5-6. After the *UnnamedRelationInformationRequest* receives an acceptable response from the user, it will rename the relation in the ER model to the recognized handwriting's text.

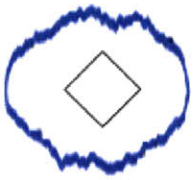


Figure 5-5: Screenshot of the multimodal question associated with an *UnnamedRelationInformationRequest* where the system asks the user "What is the name of this relation?" while identifying the unnamed relation with a blue circle.

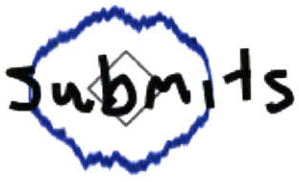


Figure 5-6: An example of acceptable user input for the *UnnamedRelationInformationRequest*.

PrimaryKeyInformationRequest

The *PrimaryKeyInformationRequest* is activated when the ER model contains an entity that has not been assigned a primary key. The multimodal question associated with this request is shown in Figure 5-1. The system expects the user to respond by drawing the letters "PK" on top of one or more of the entity's attributes. An example of an acceptable user response is shown in Figure 5-2. After the *UnnamedInformationRequest* receives an acceptable response from the user, it will assign the selected attributes to be the entity's primary key in the ER model.

Figure 5-7: Screenshot of the multimodal question associated with an *PrimaryKeyInformationRequest* where the system asks the user "Please select one or more of this entity's attributes to be its primary key" while identifying the entity with a blue circle.

Figure 5-8: An example of acceptable user input for the *PrimaryKeyInformationRequest*.

UnassignedAttributeInformationRequest

The *UnassignedAttributeInformationRequest* is activated when the ER Model contains an attribute that is not associated with an entity. The multimodal question associated with this request is shown in Figure 5-9. The system expects the user to indicate the attribute's entity by drawing a line from the attribute to the entity. An example of an acceptable user response is shown in Figure 5-10. After the *UnassignedAttributeInformationRequest* receives an acceptable response from the user, it will assign the attribute to the selected entity in the ER model.

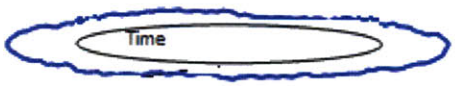


Figure 5-9: Screenshot of the multimodal question associated with an *UnassignedAttibuteInformationRequest* where the system asks the user "Which entity has this attribute?" while identifying the attribute with a blue circle.

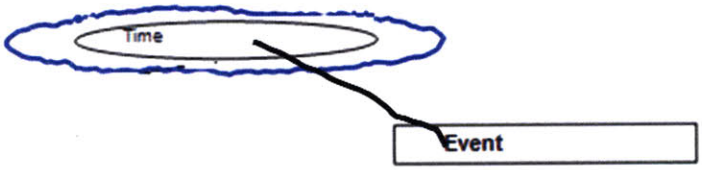


Figure 5-10: An example of acceptable user input for the *UnassignedAttributeInformationRequest*.

UnassociatedRelationInformationRequest

The *UnassociatedRelationInformationRequest* is activated when the ER Model contains a relation that is not associated with two entities. The multimodal question associated with this request is illustrated in Figure 5-11. The system expects the user to identify one of the relation's entities by drawing a line from the relation to that entity. An example of an acceptable user response is shown in Figure 5-12. After the *UnassignedAttributeInformationRequest* receives an acceptable response from the user, it will assign the selected entity to the relation in the ER model.

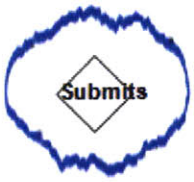


Figure 5-11: Screenshot of the multimodal question associated with an *UnassociatedRelationInformationRequest* where the system asks the user "Which entities are assigned to this relation?" while identifying the relation with a blue circle.

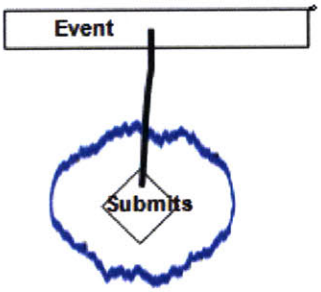


Figure 5-12: An example of acceptable user input for the *UnassociatedRelationInformationRequest*.

RelationCardinalityInformationRequest

The *RelationCardinalityInformationRequest* is activated when the ER model contains a relation that has not been assigned a cardinality. In this case, the *RelationCardinalityInformationRequest* asks the user the following yes/no questions in order to determine whether the relation's

cardinality is many-to-many, many-to-one, or one-one (*entity1* and *entity2* refer to the two entities that are related by this relation):

Question 1: "Can *entity1* be associated with more than one *entity2*?"
Question 2: "Can *entity2* be associated with more than one *entity1*?"

Table 5-2 illustrates how the system identifies the cardinality of the relation based on the responses given to Questions 1 and 2.

Table 5-2: Illustration as to how the system determines the relation's cardinality based on the user's response to the Questions 1 and 2

User Response to Question 1	User Response to Question 2	Relation's Cardinality
Yes	Yes	Many-to-many
Yes	No	Many-to-one
No	Yes	One-to-many
No	No	One-to-many

The *RelationCardinalityInformationRequest* also allows the user to input each side of the cardinality by drawing on the ER diagram. Specifically, if the user writes a "1" on the unknown of a cardinality, the system will identify that side of the cardinality as being "one." Conversely, if the user writes a "1-N", the system will identify that side as being "many." Figure 5-13 illustrates how the user can specify a one-to-many cardinality for the *occurs on* relation based on this method.

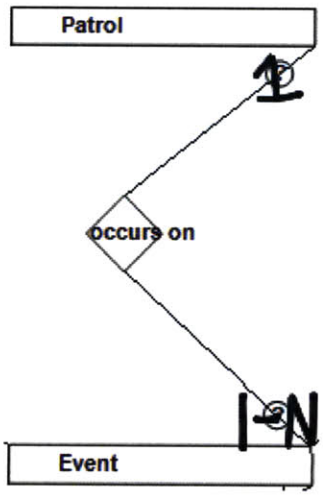


Figure 5-13: An example of acceptable user input which specifies a one-to-many cardinality for the *occurs on* relation.

After the *RelationCardinalityInformationRequest* determines the cardinality of the relation, it updates the ER model accordingly.

DrawingInformationRequest

The *DrawingInformationRequest* is an *InformationRequest* that is activated at all times, runs in the background, and does not interrupt the user with a question. The purpose of this class is to recognize when the user wants to add a new entity, attribute, or relation to the entity-relationship diagram. The expected sketch input for this class includes a rectangle corresponding to a new entity (Figure 5-14a), an ellipse corresponding to a new attribute (Figure 5-14b), and a diamond corresponding to a new relation (Figure 5-14c). After the *DrawingInformationRequest* identifies that the user has drawn a rectangle, it will add a new, unnamed entity to the ER model, and will do likewise if an ellipse (new attribute) or a diamond (new relation) is drawn.

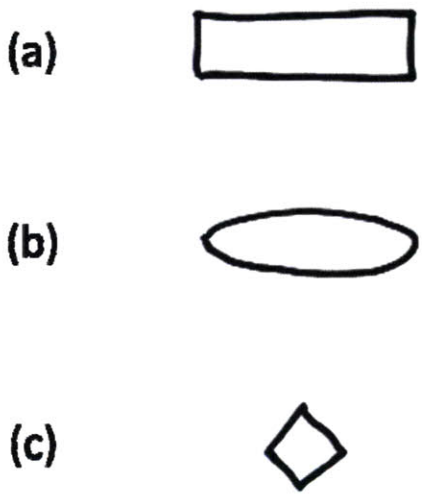


Figure 5-14: The expected input for (a) a new entity, (b) a new attribute, and (c) a new relation.

RedundancyInformationRequest

The *RedundancyInformationRequest* is activated when the system has analyzed the data stored in the relational database and discovered a new possible FD candidate. In this case, the *RedundancyInformationRequest* asks the user the following yes/no question in order to determine whether the FD candidate ($c \rightarrow c^*$) represents an actual FD in the database (c is

composed of the attributes, $c_1, c_2, \ldots$ and c_N, and c^* is composed of the attributes, $c^*_1, c^*_2, \ldots,$ and c^*_M):

Question: "For a given $c_1, c_2, \ldots$ and c_N is/are the $c^*_1, c^*_2, \ldots,$ and c^*_M always the same?"

For example, the question generated for the FD, AB → C is: "For a given A and B is the C always the same?".

If the user responds "No", then the FD candidate does not represent an actual FD in the database. However, if the user responds "Yes", the FD does represent an actual FD, and the system utilizes the normalization algorithm described in Chapter 4.4 to reformat the database design.

5.2.3 Interaction Walkthrough

The sequence of screenshots in this section illustrates how MIDOS can be used to create and normalize a database for an after-action report wiki. The user's strokes are colored black and the system's strokes are colored blue, with both the user's and the system's speech being displayed in the bottom of the interface.

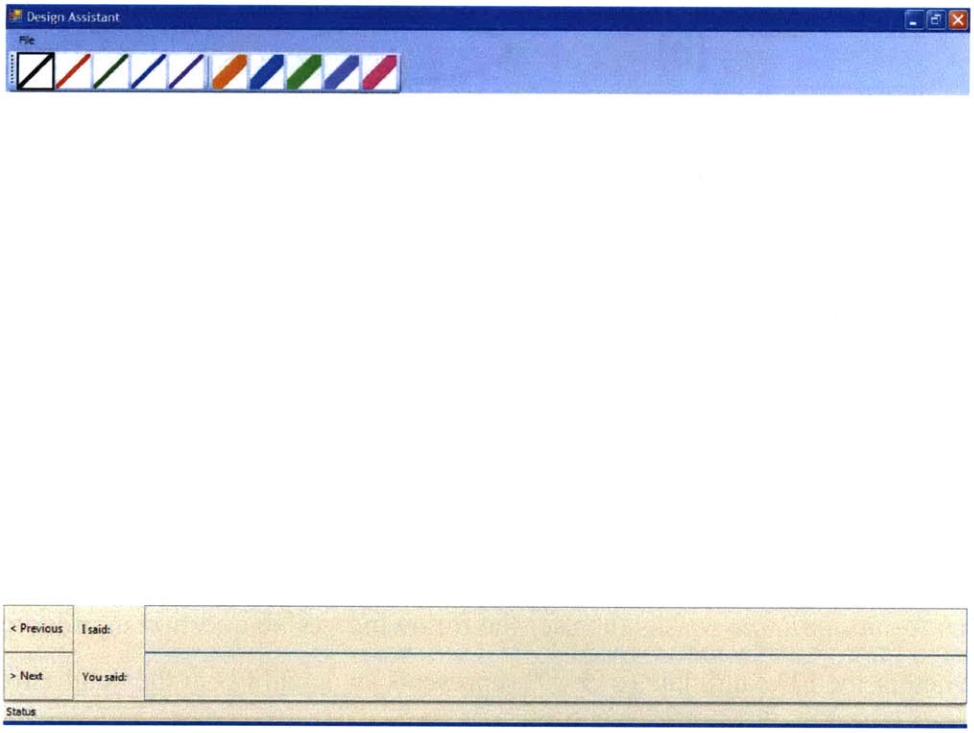


Figure 5-15: The starting state of the entity-relationship diagram.

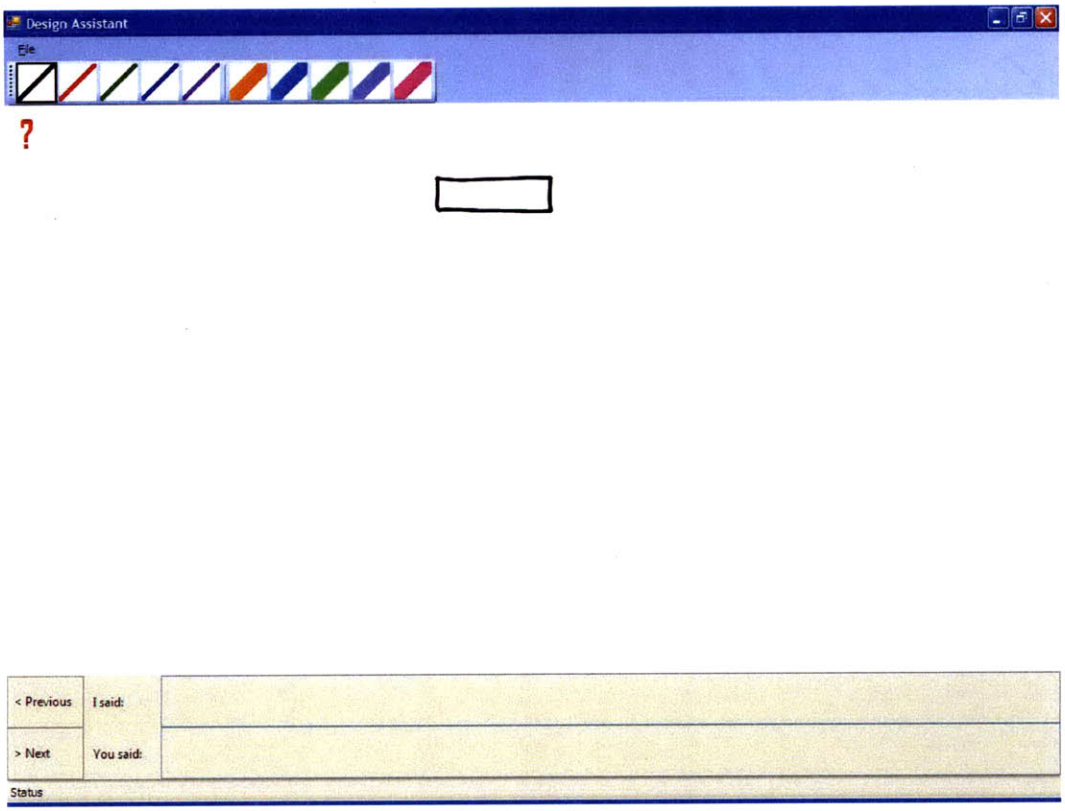


Figure 5-16: The user draws a new entity.

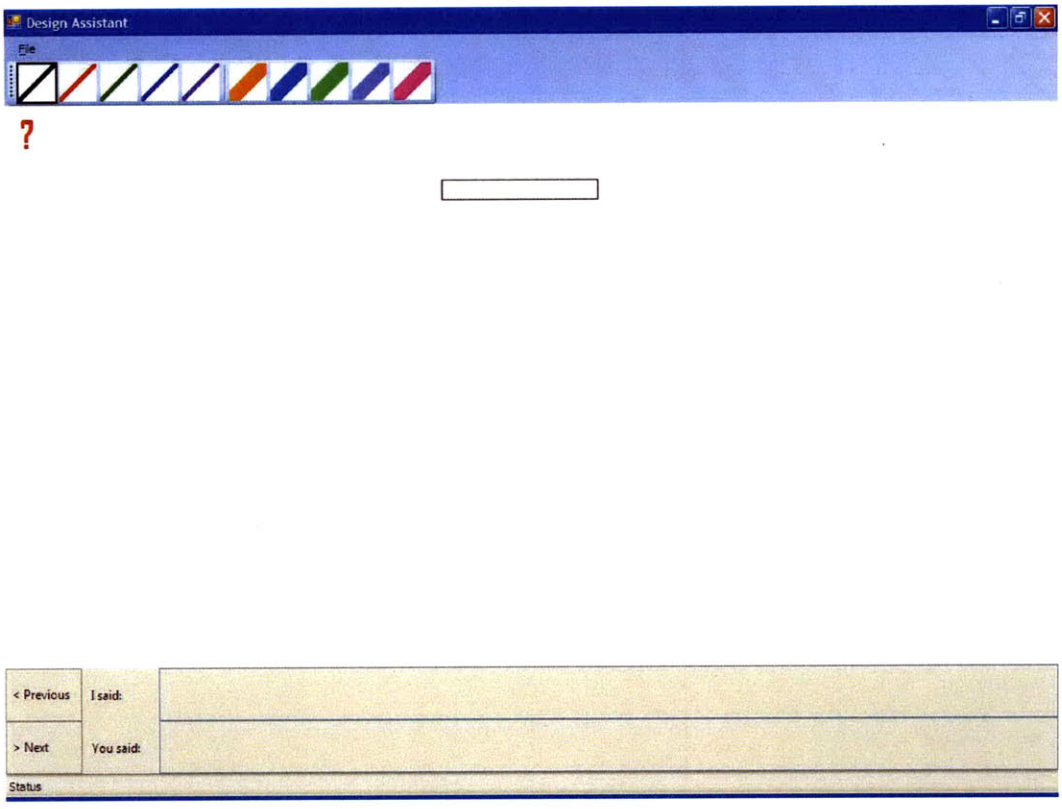


Figure 5-17: The system recognizes the user's input and adds a new entity to the entity-relationship diagram.

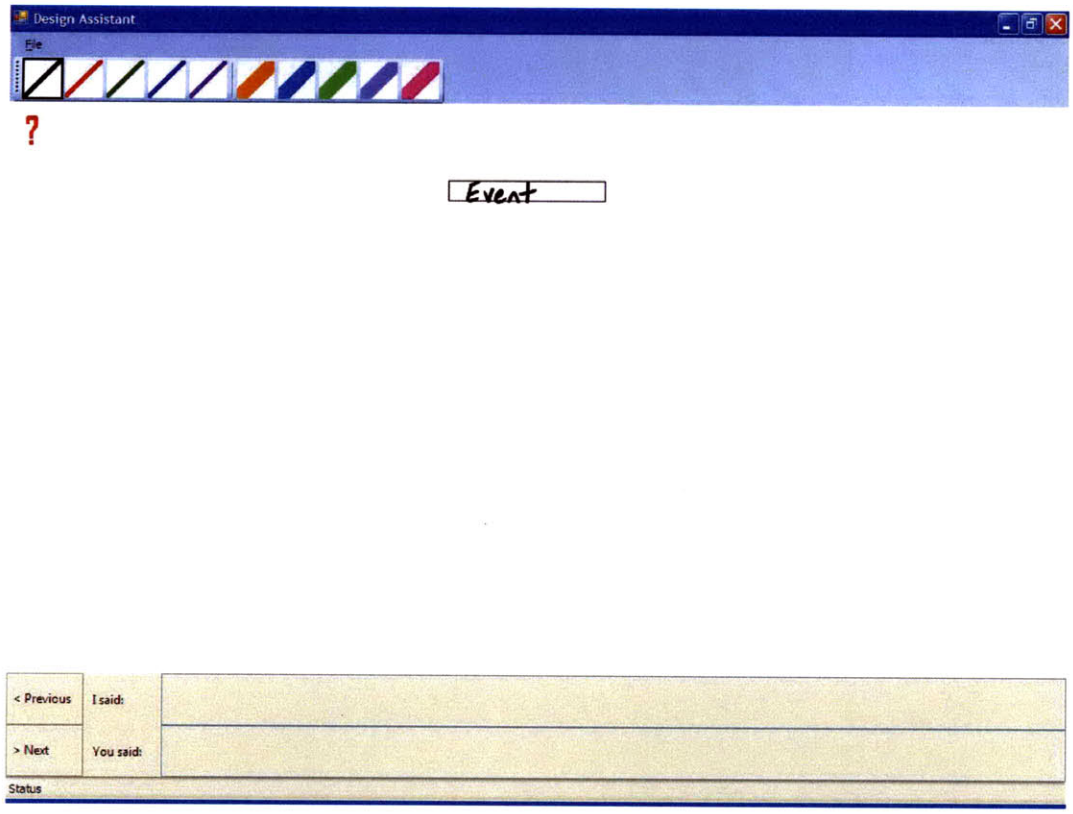


Figure 5-18: The user writes in a name for the new entity.

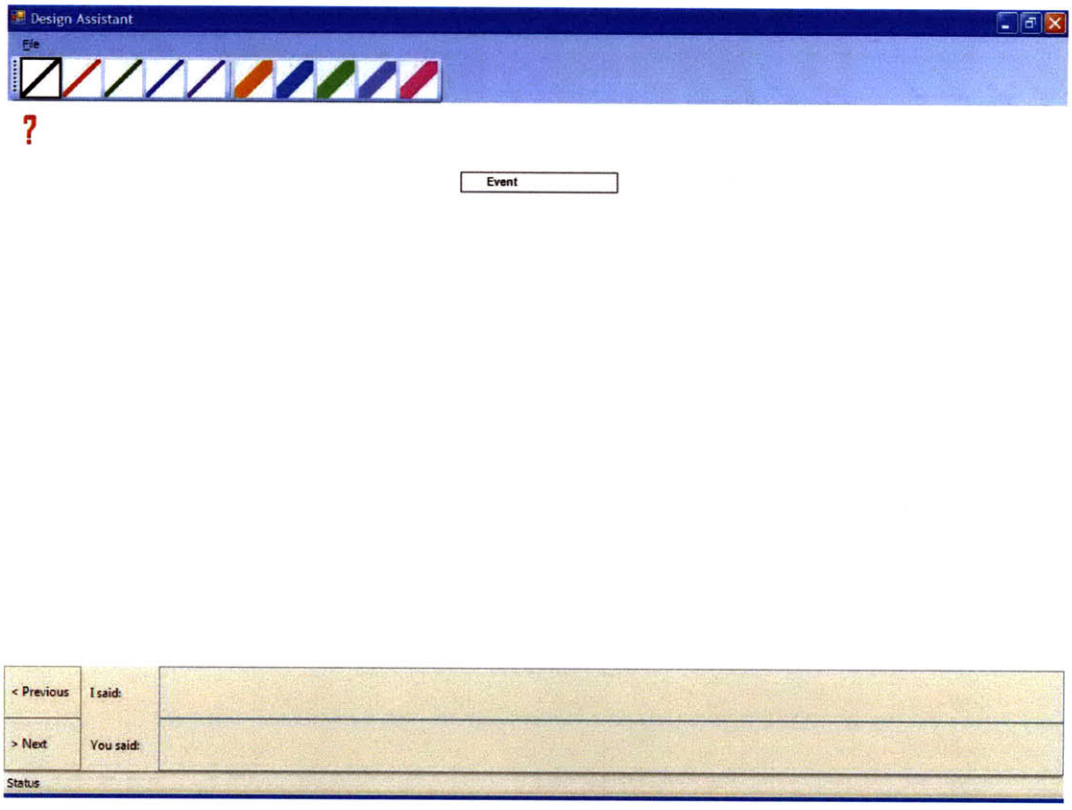


Figure 5-19: The system recognizes the user's handwriting and renames the entity.

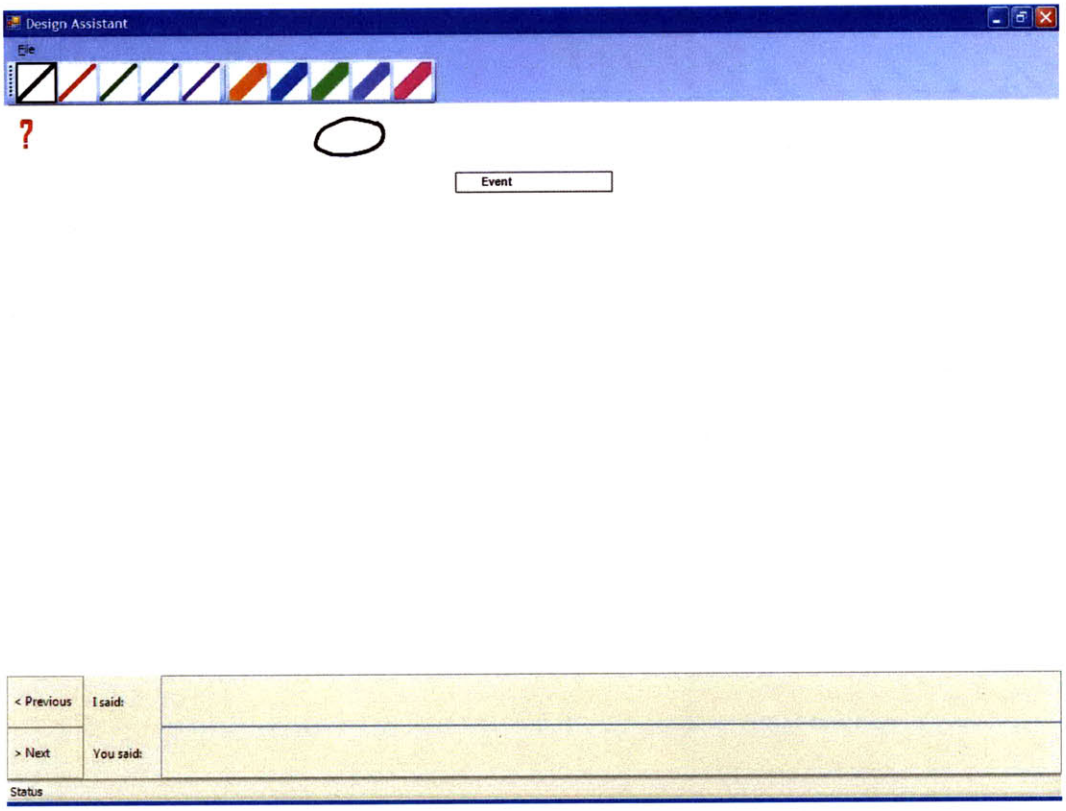


Figure 5-20: The user draws a new attribute.

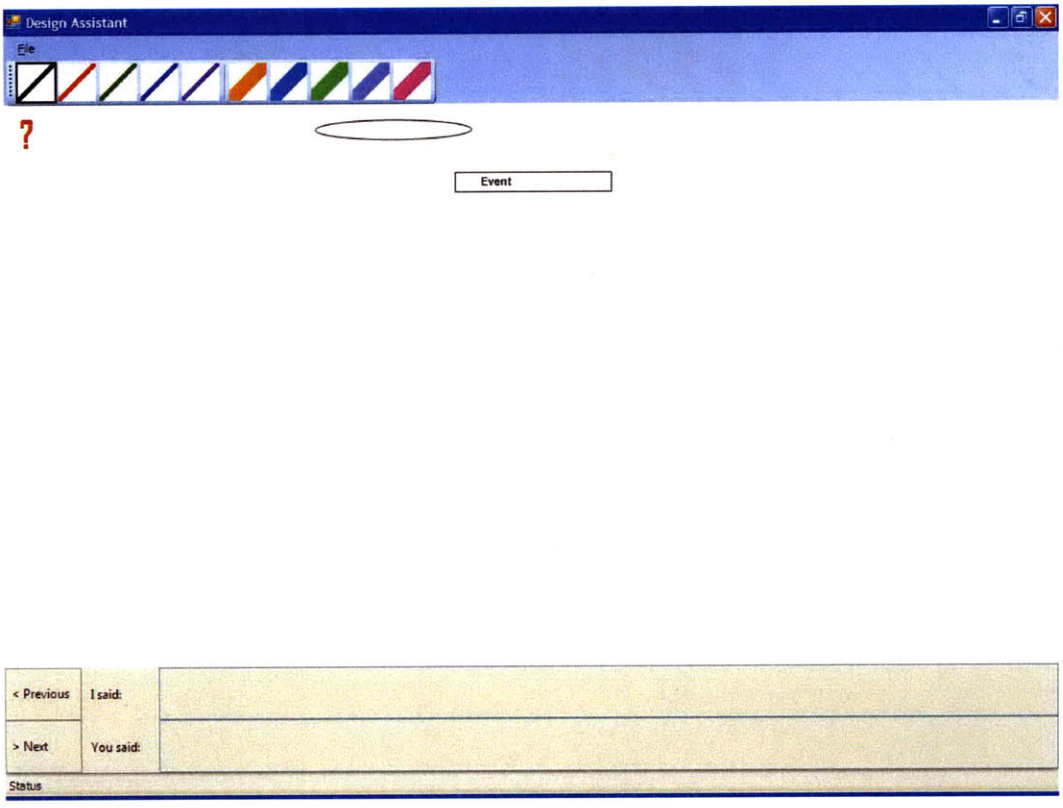


Figure 5-21: The system recognizes the user's input and adds a new attribute to the entity-relationship diagram.

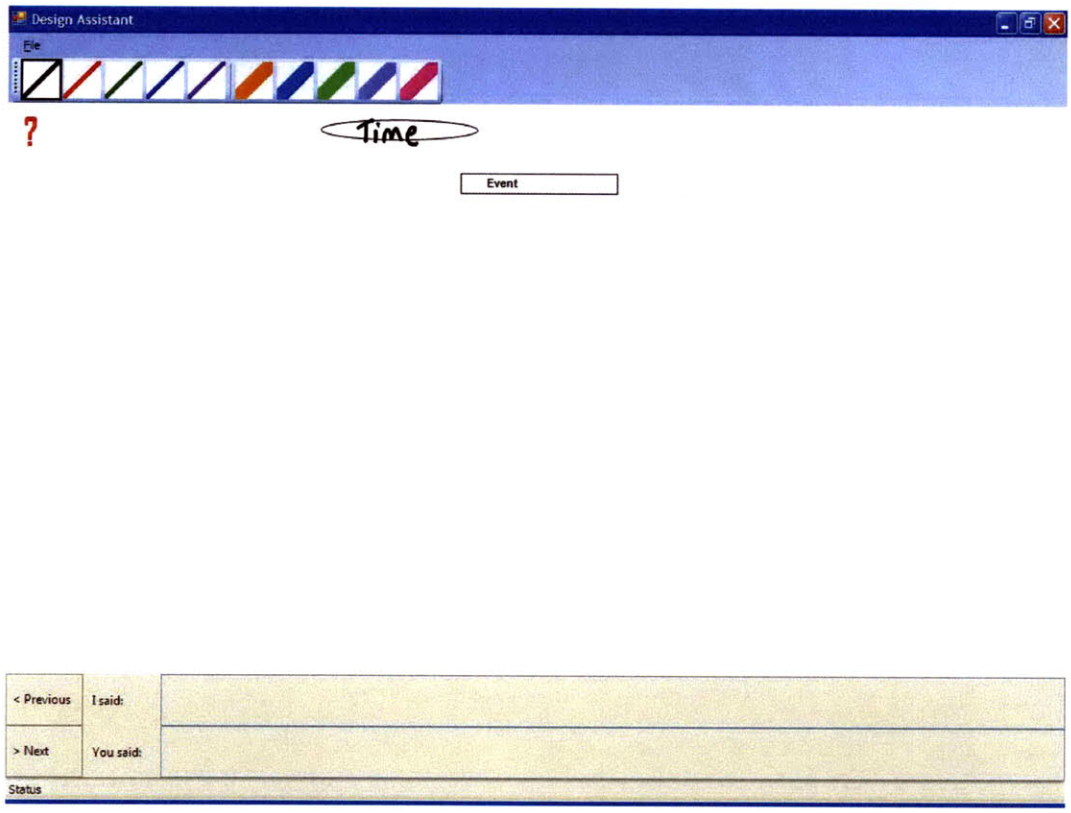


Figure 5-22: The user writes in a name for the new attribute.

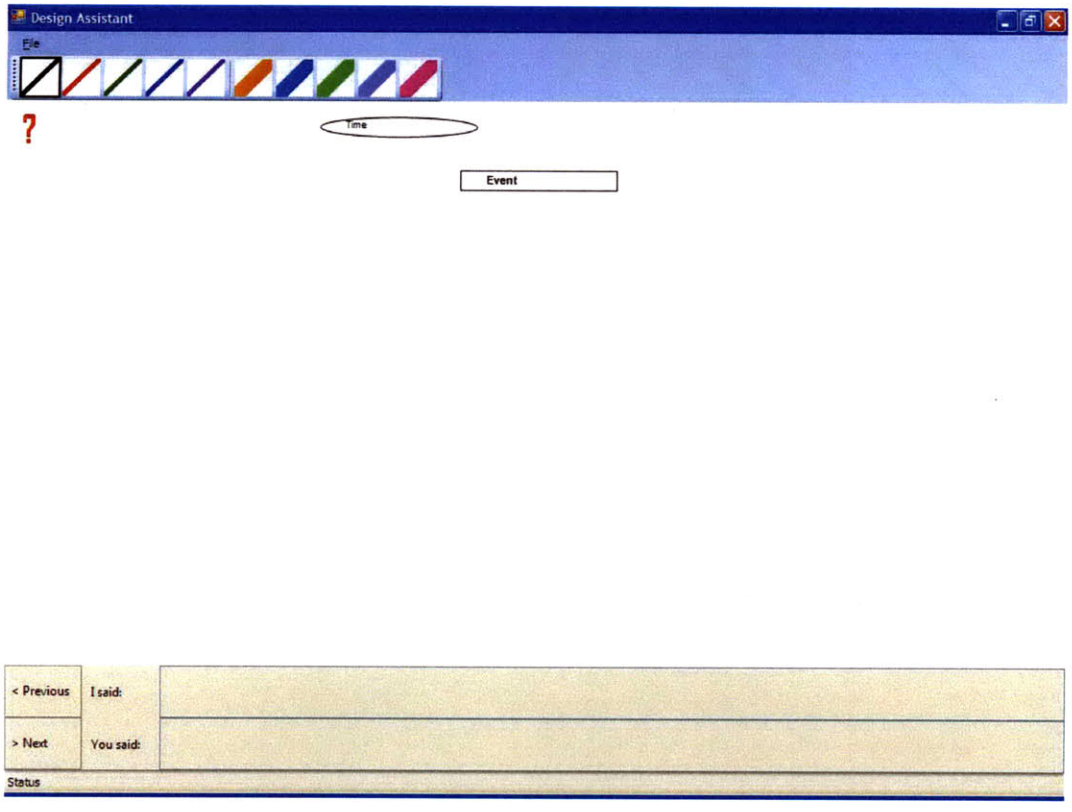


Figure 5-23: The system recognizes the user's handwriting and renames the attribute.

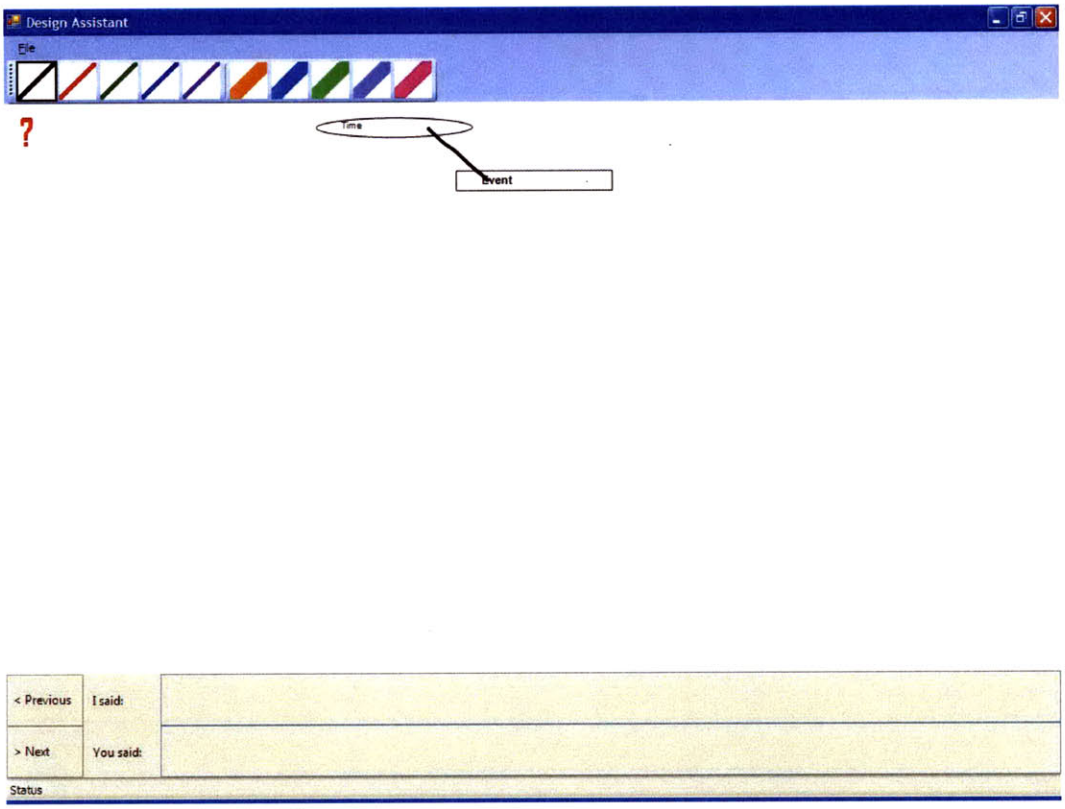


Figure 5-24: The user assigns the *Time* attribute to the *Event* entity.

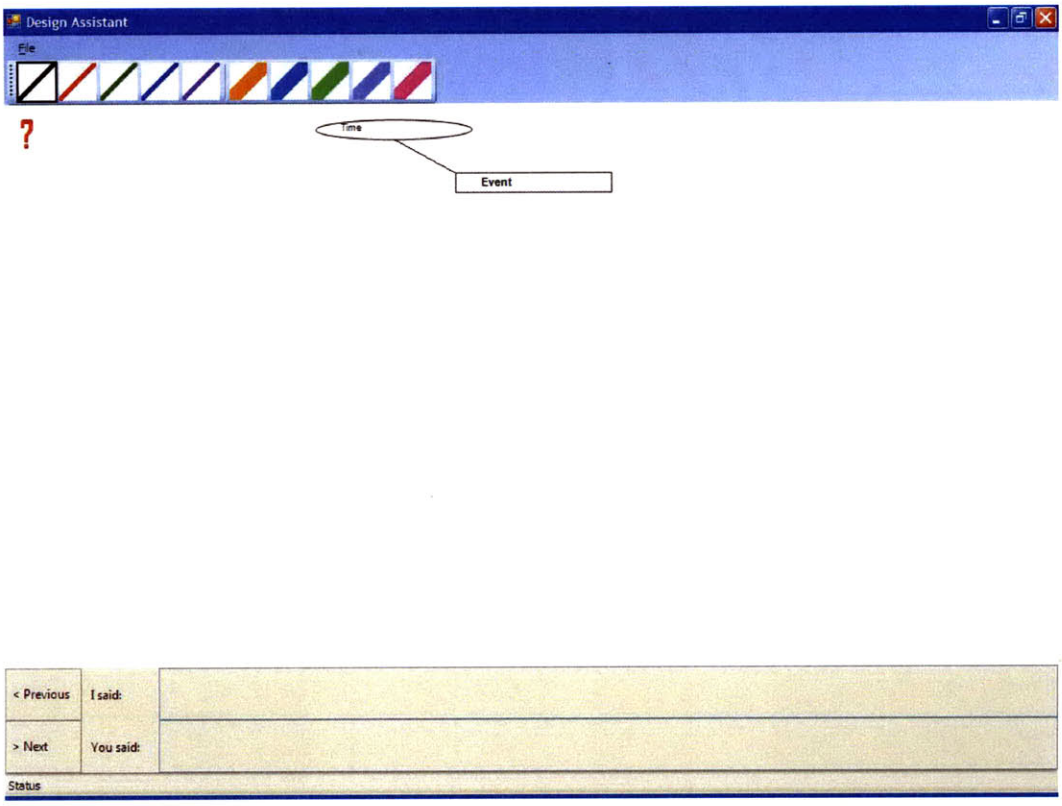


Figure 5-25: The system recognizes the user's input and assigns the *Time* attribute to the *Event* entity.

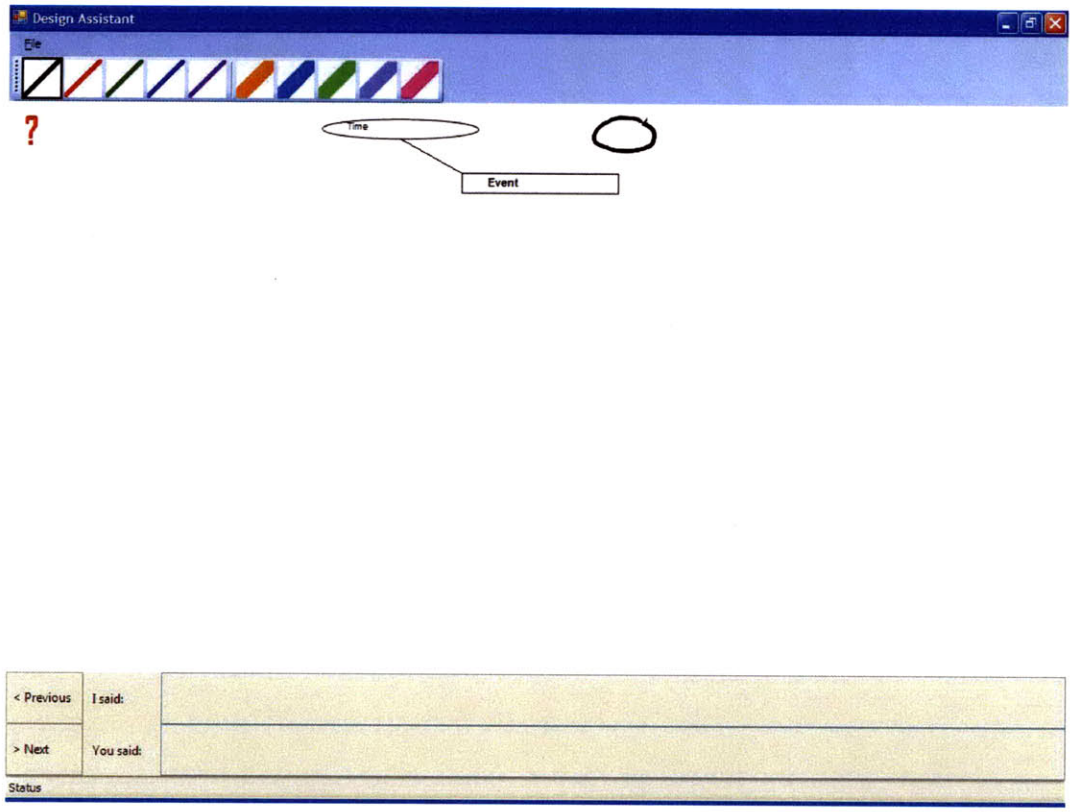


Figure 5-26: The user draws a new attribute.

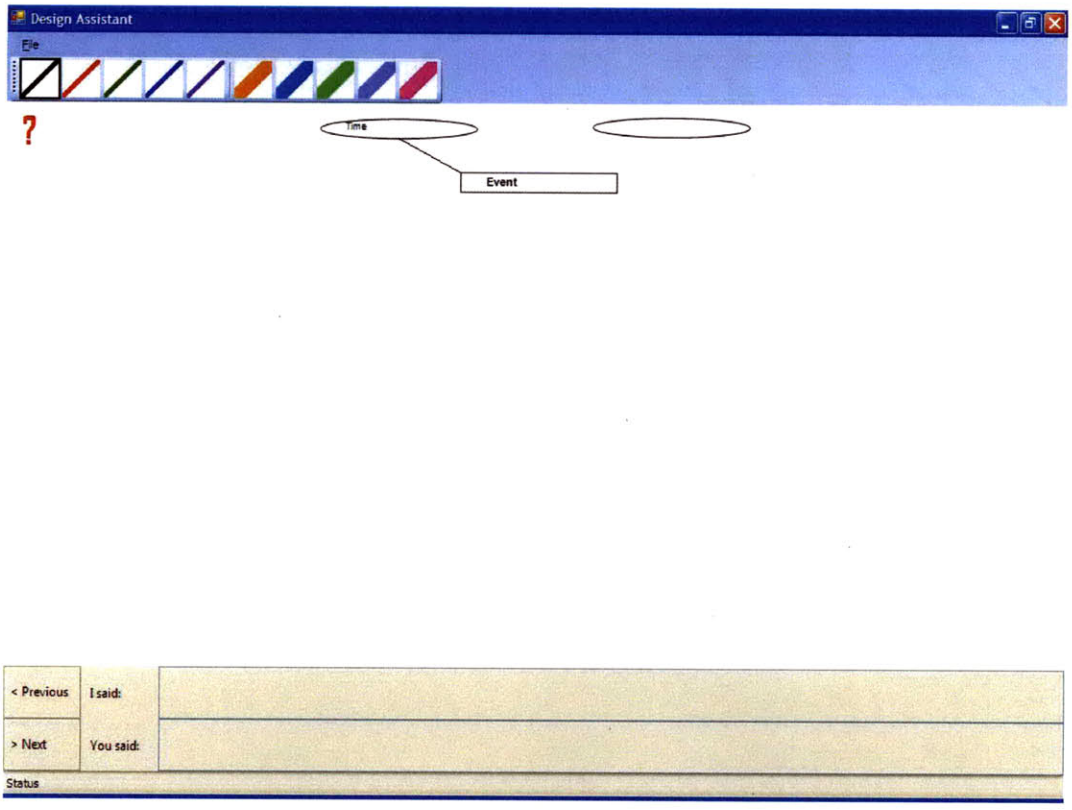


Figure 5-27: The system recognizes the user's input and adds a new attribute to the entity-relationship diagram.

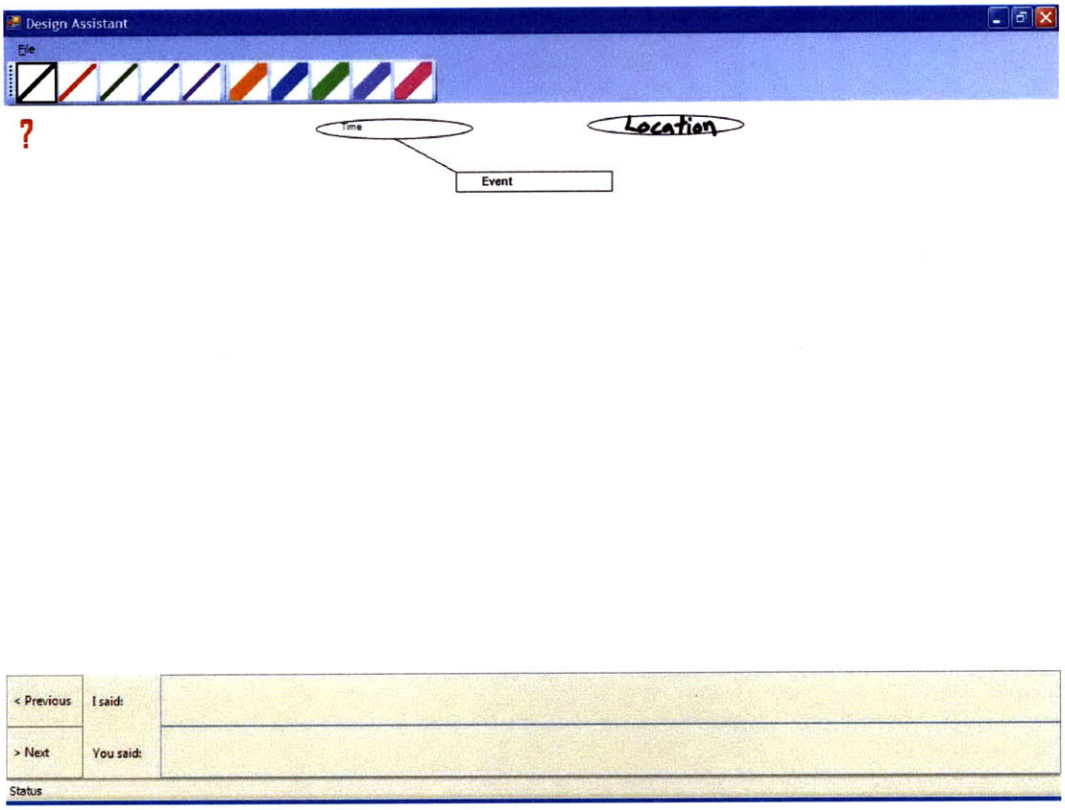


Figure 5-28: The user writes in a name for the new attribute.

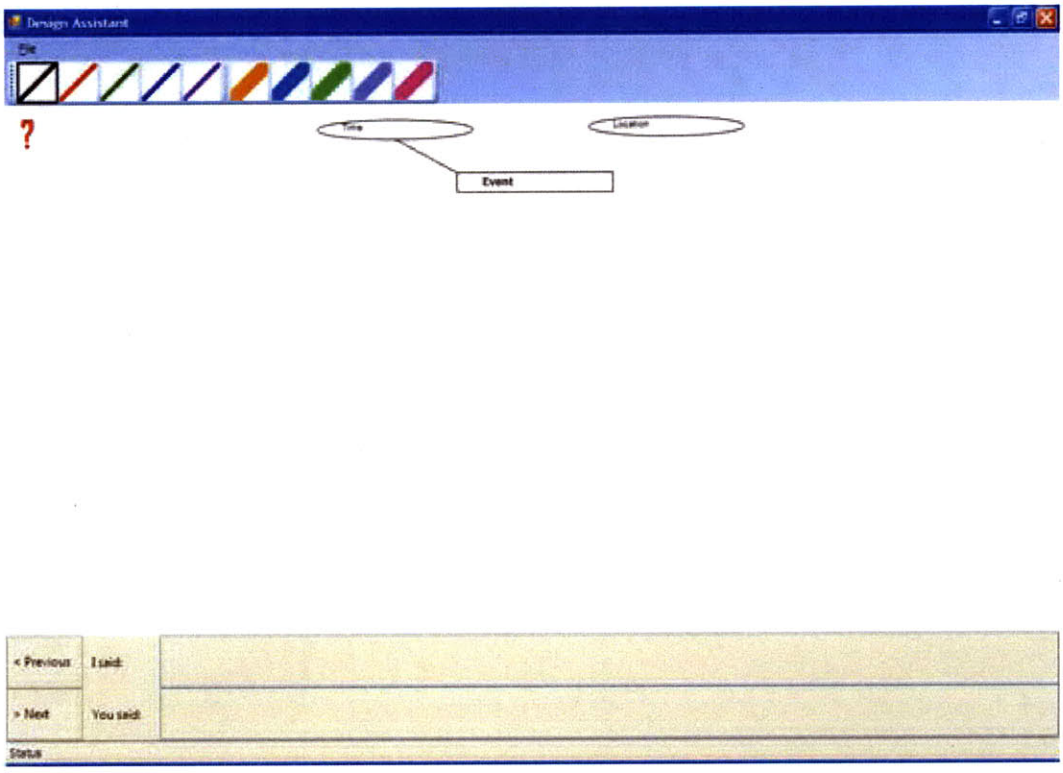


Figure 5-29: The system recognizes the user's handwriting and renames the attribute.

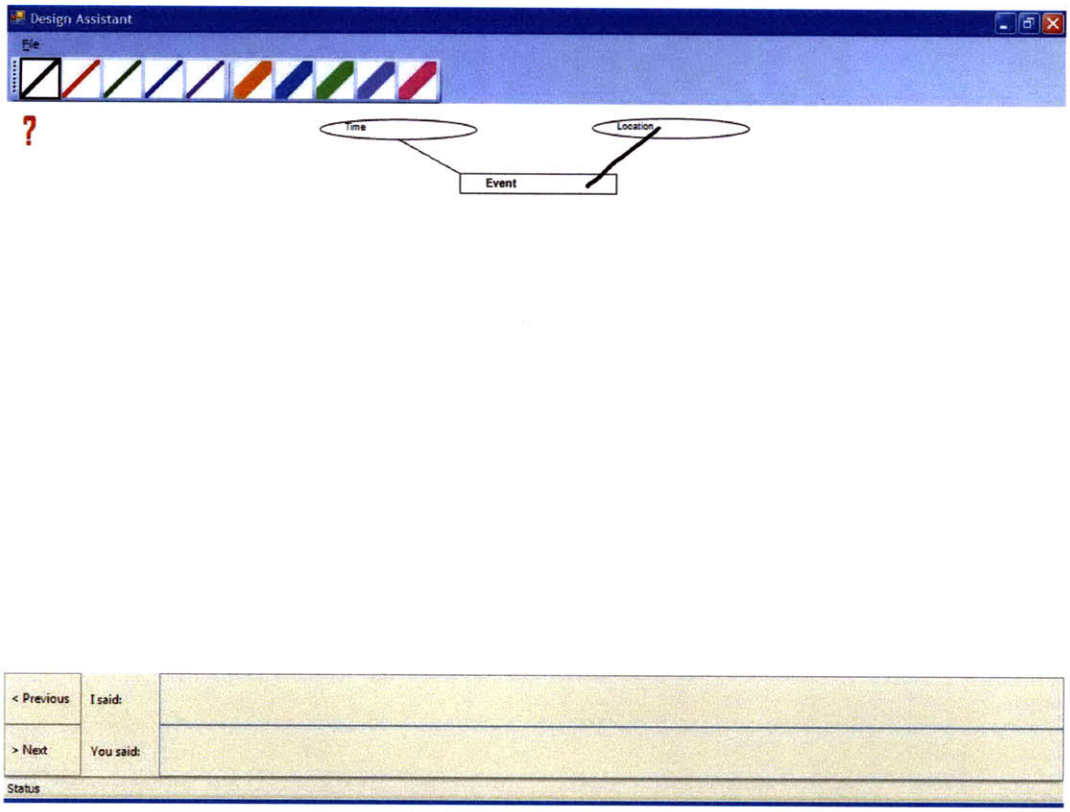


Figure 5-30: The user assigns the *Location* attribute to the *Event* entity.

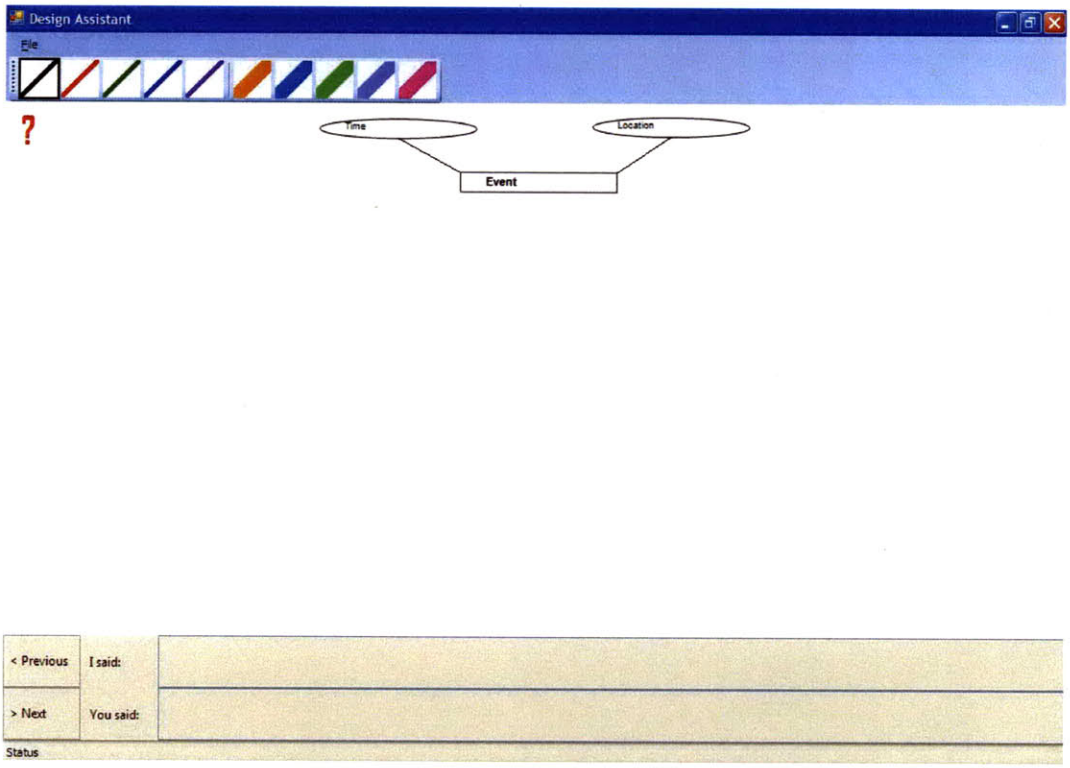


Figure 5-31: The system recognizes the user's input and assigns the *Location* attribute to the *Event* entity.

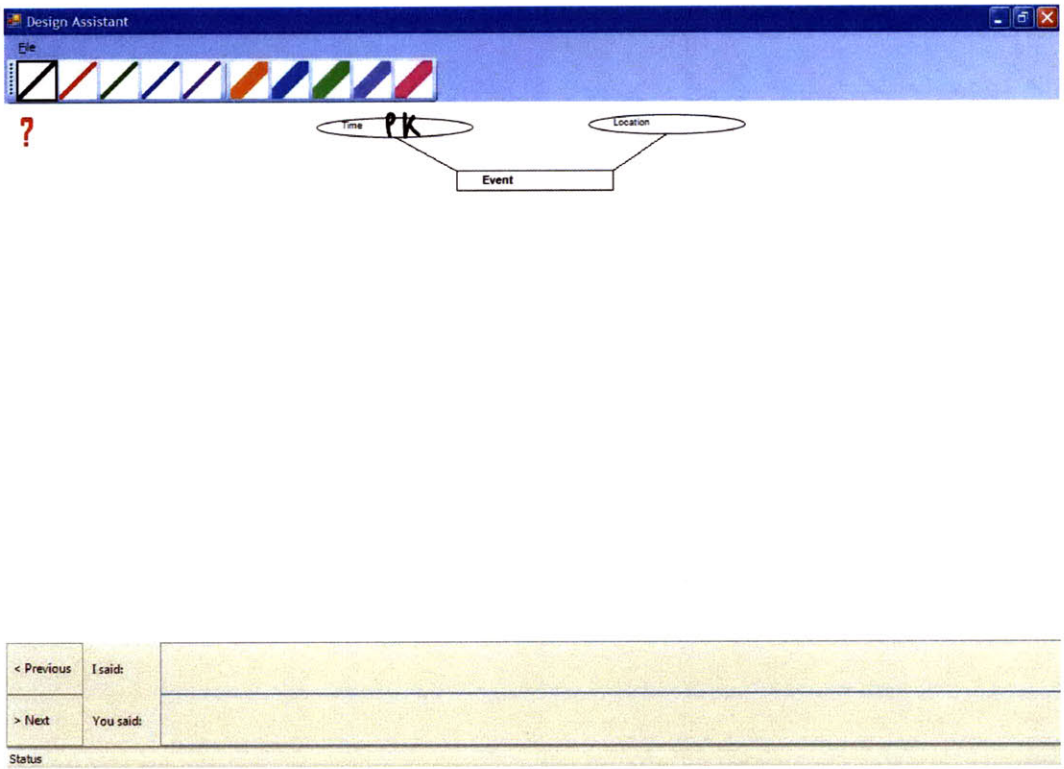


Figure 5-32: The user assigns the *Time* attribute as the primary key for the *Event* entity.

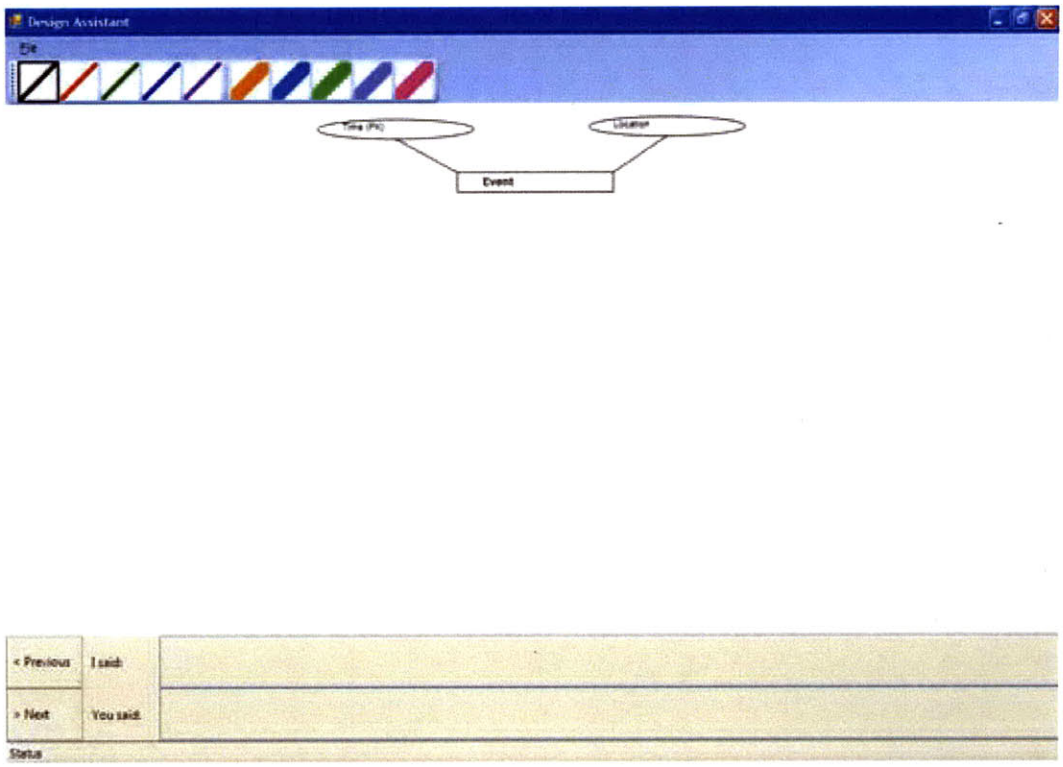


Figure 5-33: The system recognizes the user's input and assigns the *Time* attribute as the primary key for the *Event* entity.

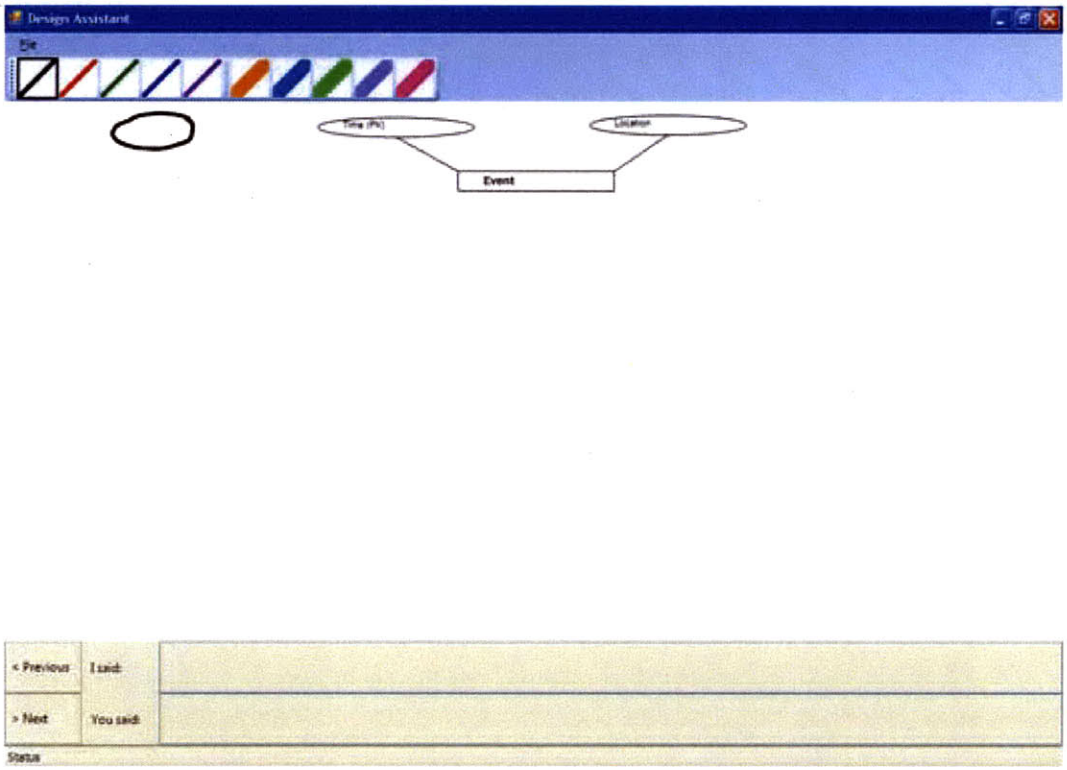


Figure 5-34: The user draws a new attribute.

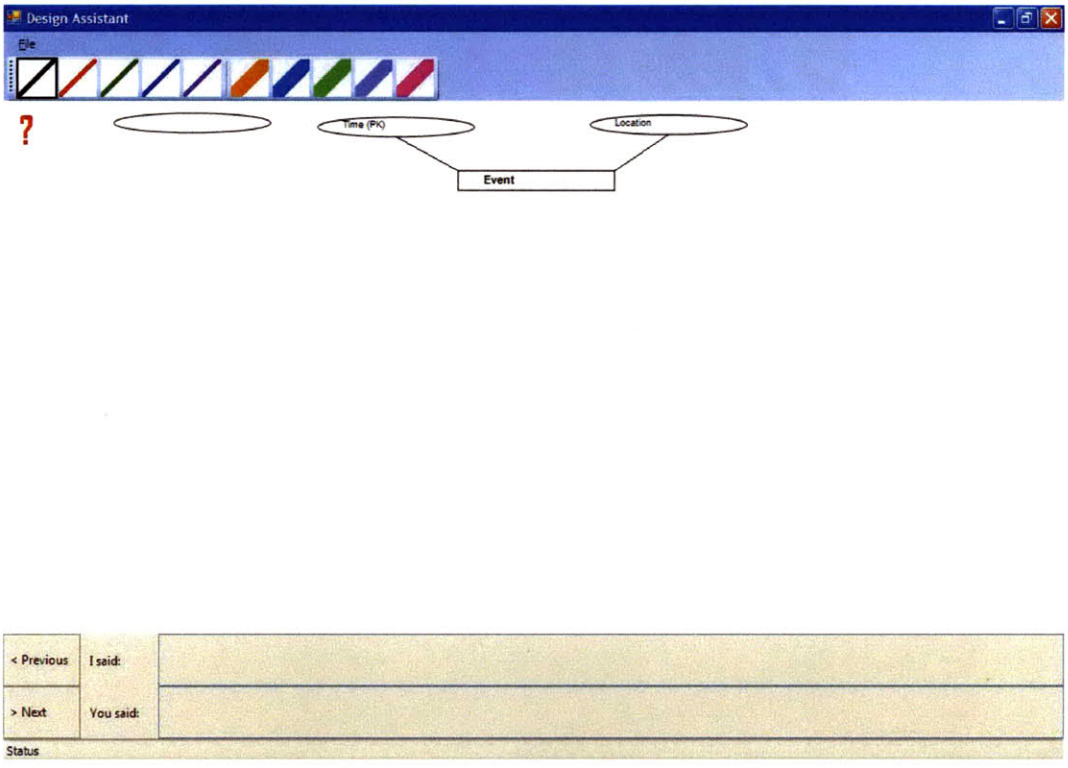


Figure 5-35: The system recognizes the user's input and adds a new attribute to the entity-relationship diagram.

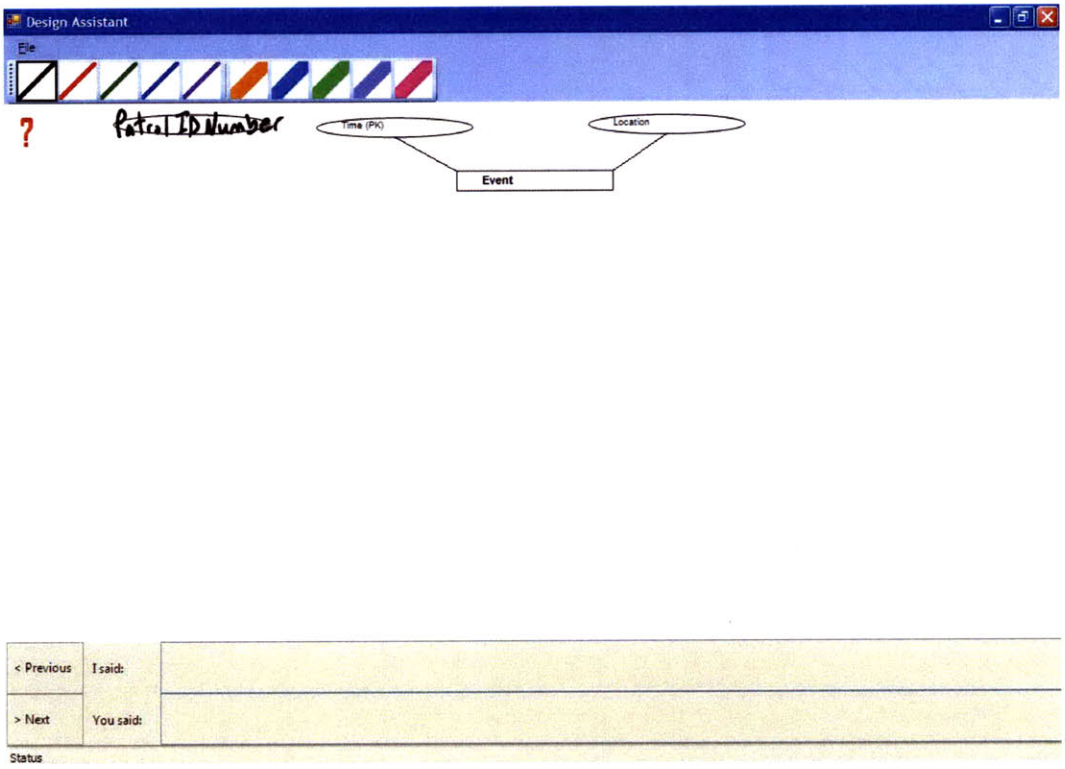


Figure 5-36: The user writes in the name for the new attribute.

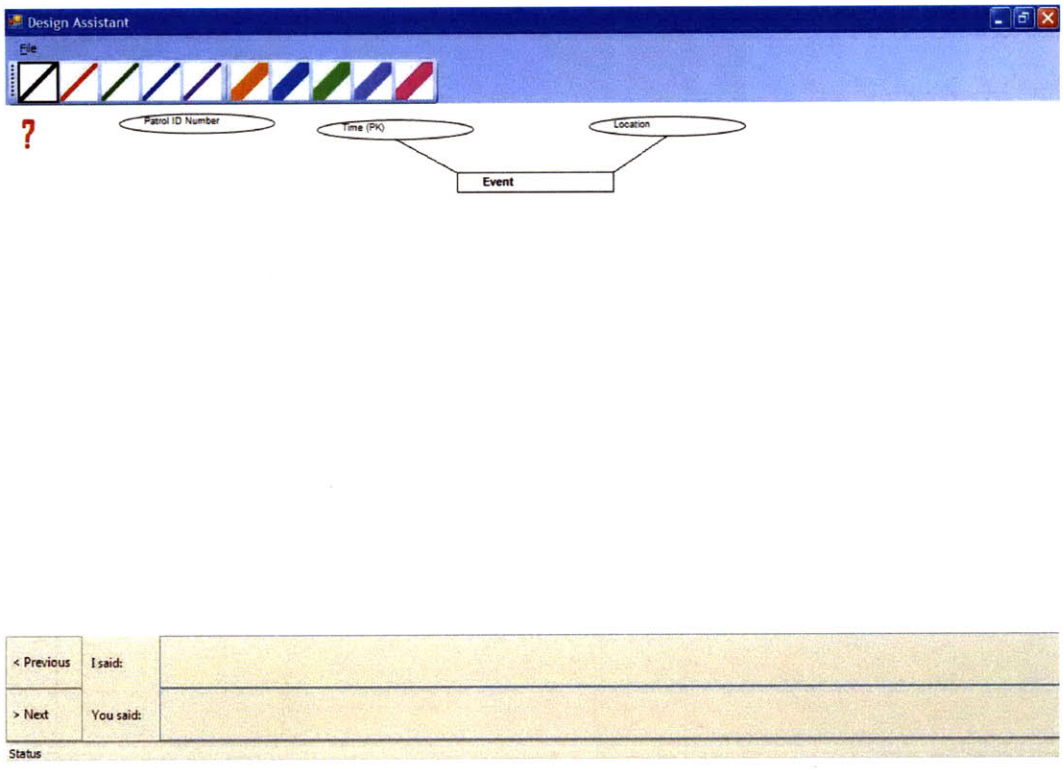


Figure 5-37: The system recognizes the user's handwriting and renames the attribute.

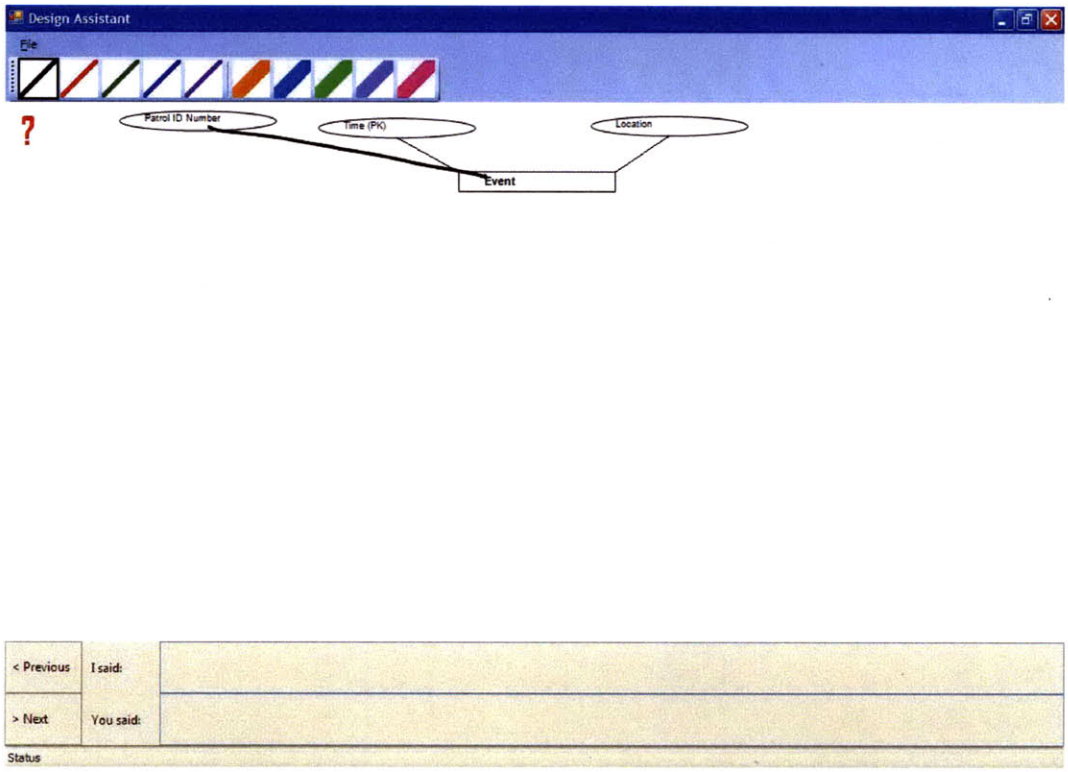


Figure 5-38: The user assigns the *Patrol ID Number* attribute to the *Event* entity.

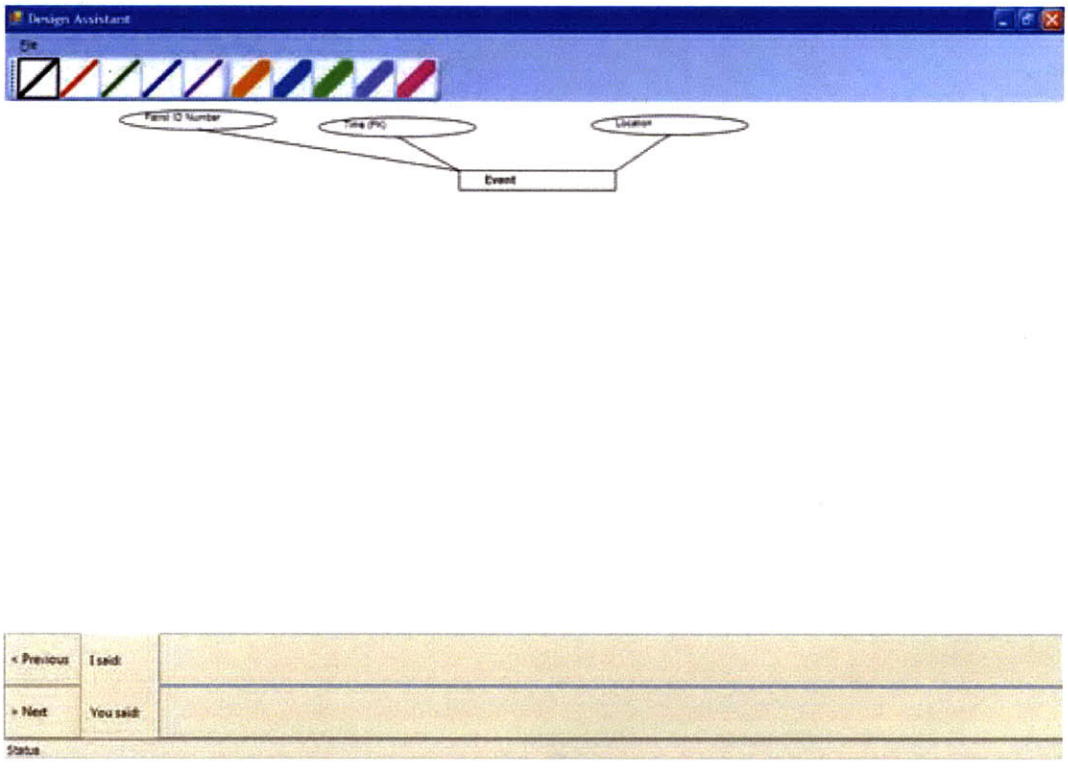


Figure 5-39: The system recognizes the user's input and assigns the *Patrol ID Number* attribute to the *Event* entity.

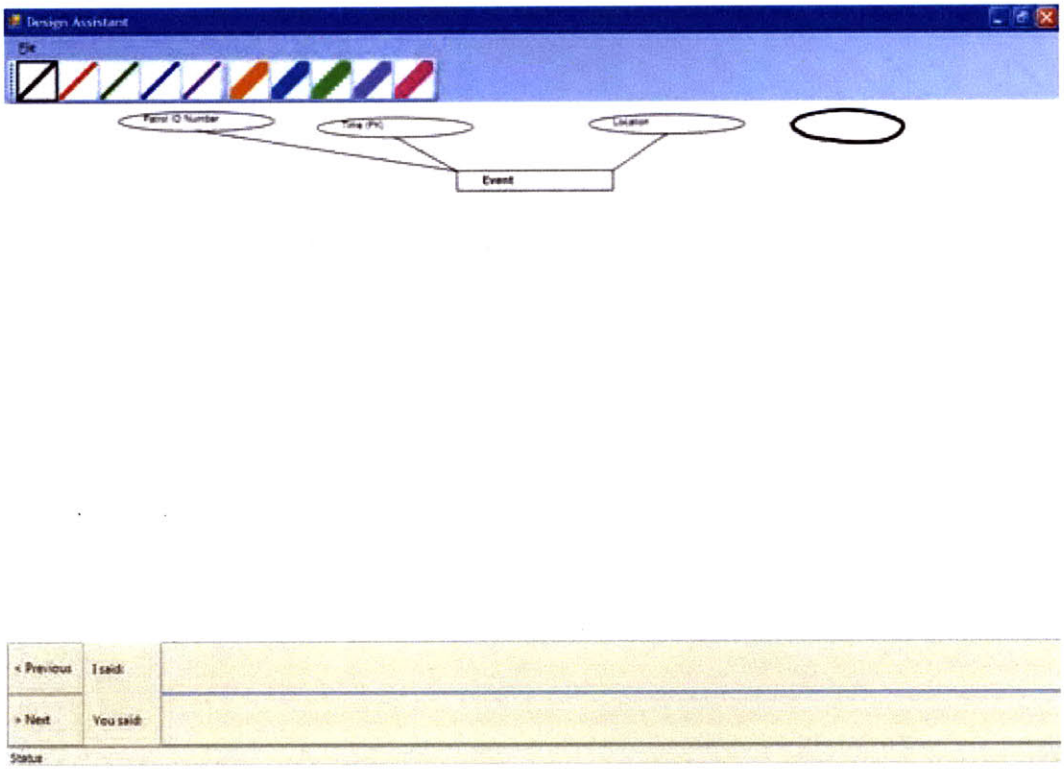


Figure 5-40: The user draws a new attribute.

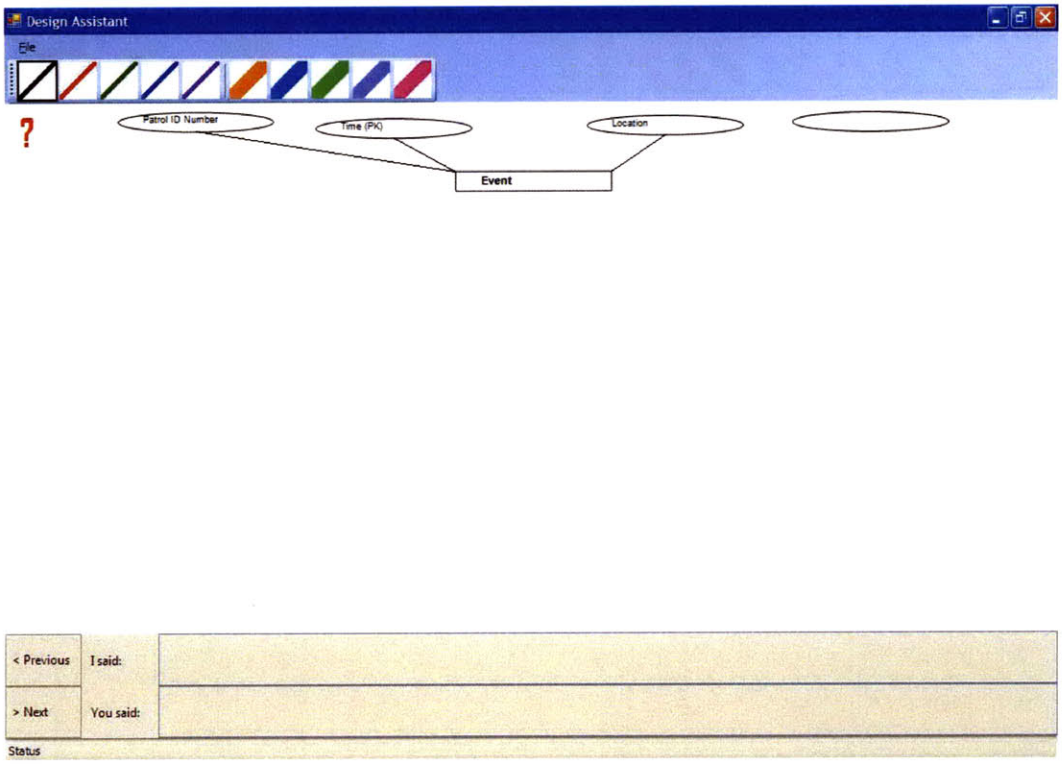


Figure 5-41: The system recognizes the user's input and adds a new attribute to the entity-relationship diagram.

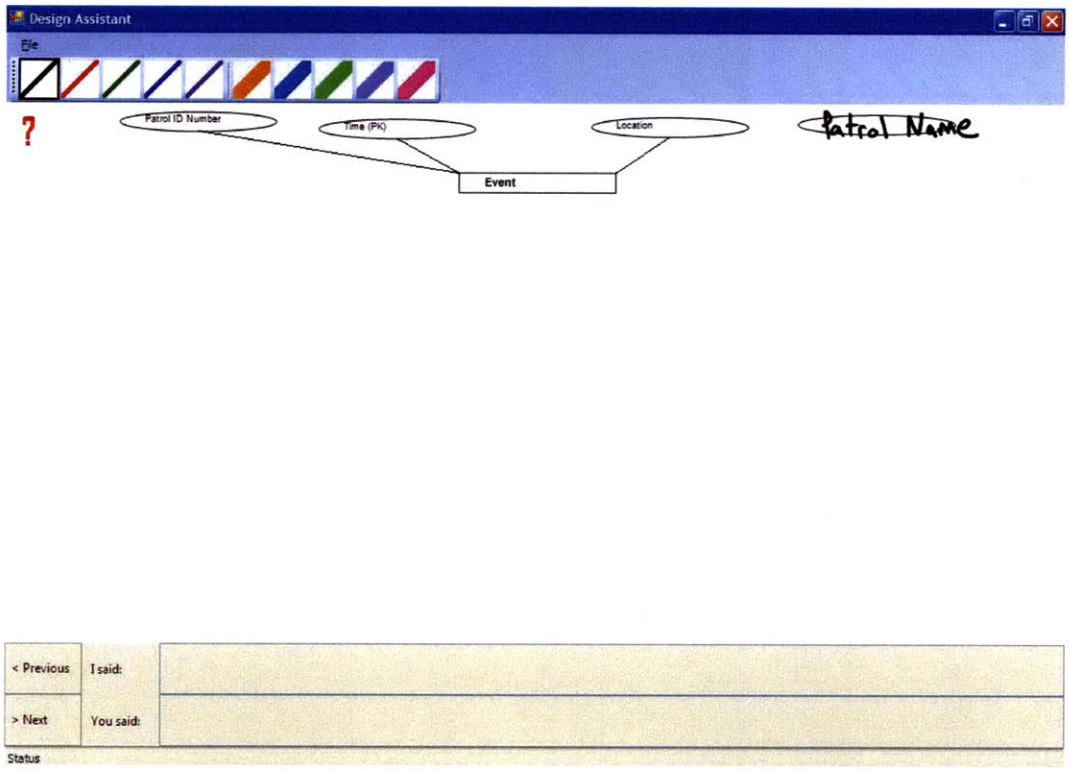


Figure 5-42: The user write in the name for the new attribute.

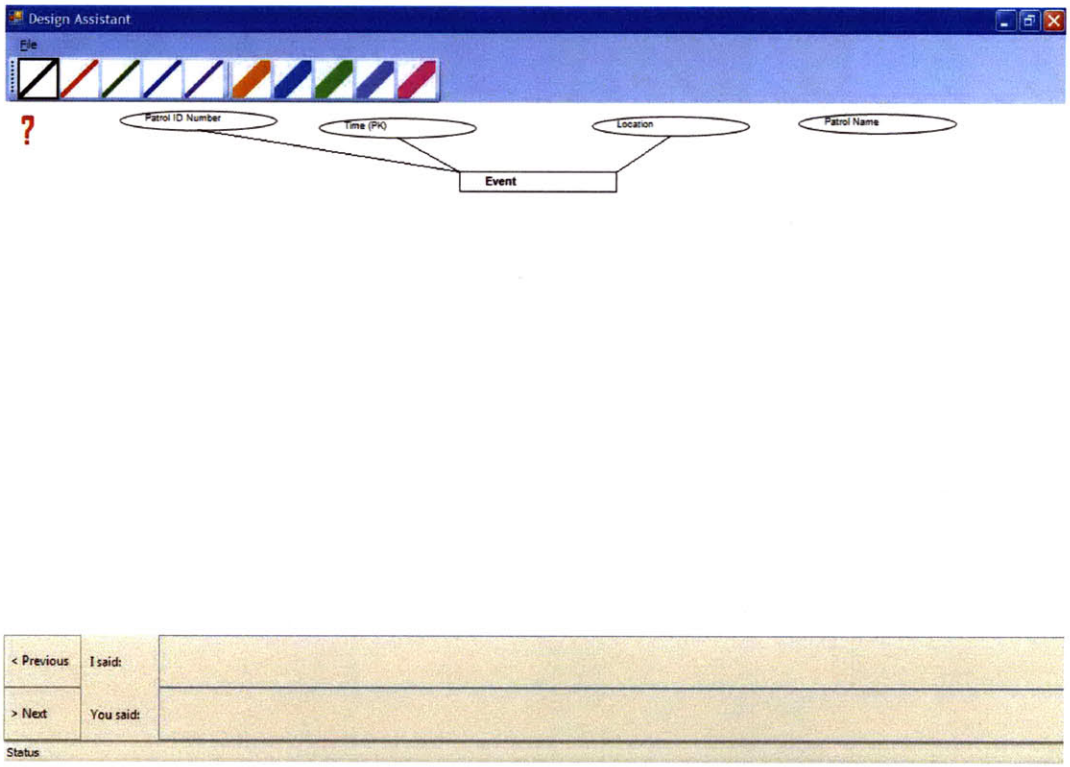


Figure 5-43: The system recognizes the user's handwriting and renames the attribute.

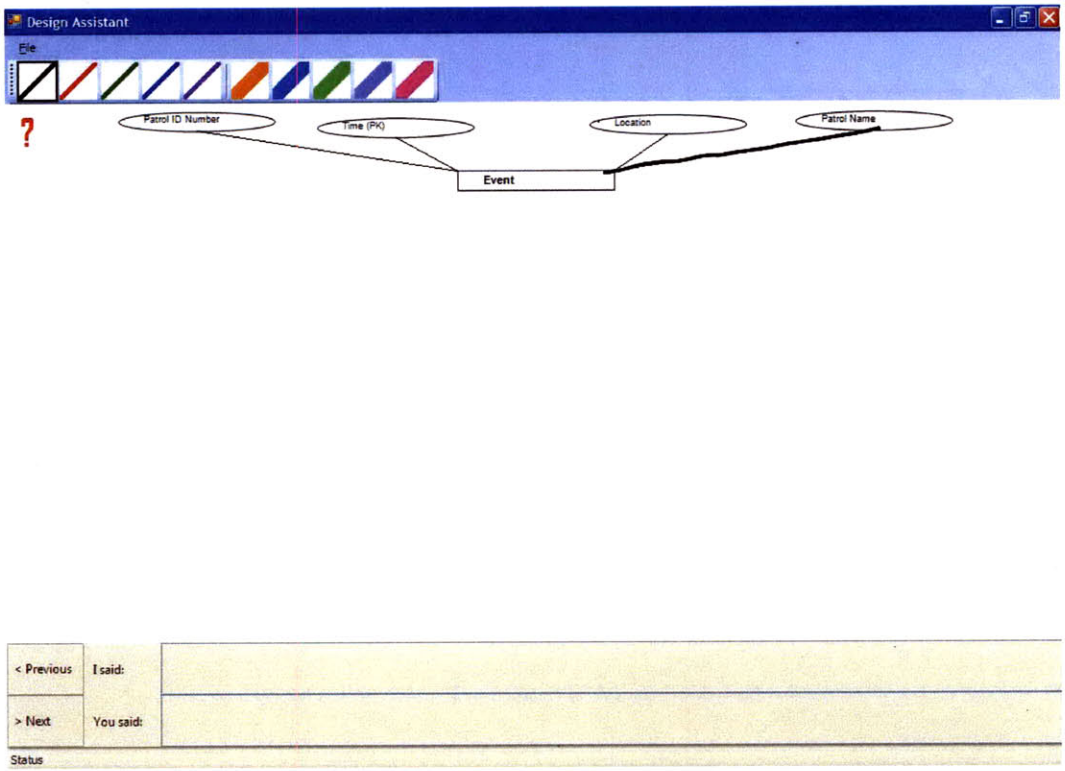


Figure 5-44: The user assigns the *Patrol Name* attribute to the *Event* entity.

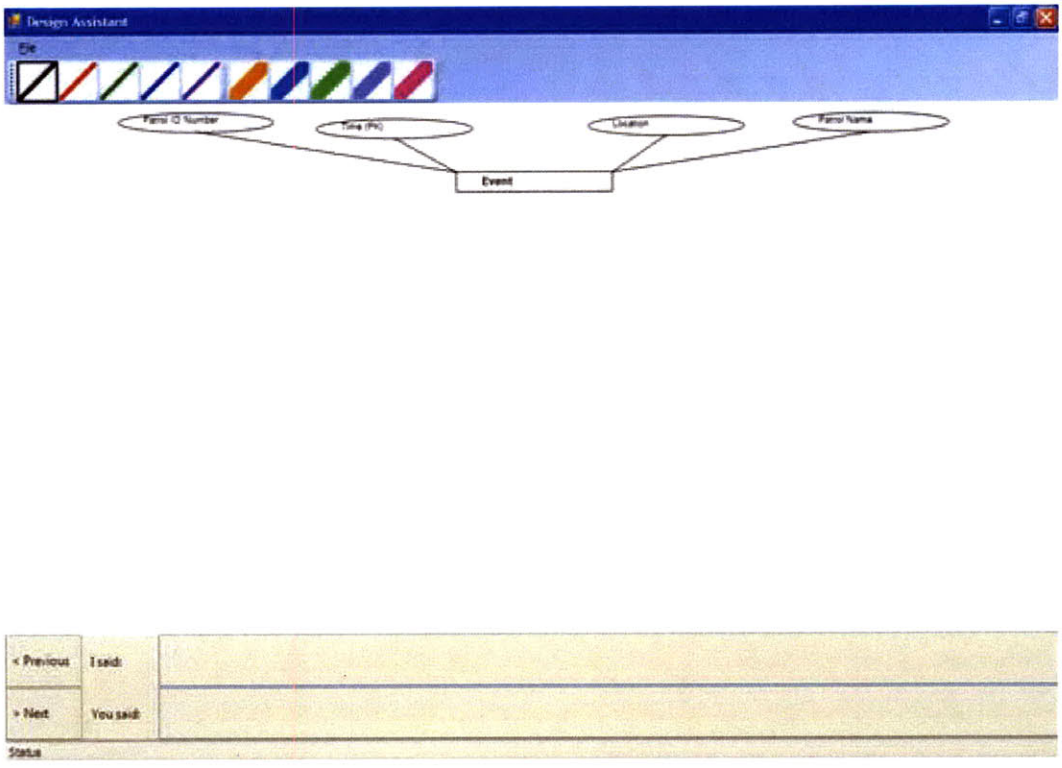


Figure 5-45: The system recognizes the user's input and assigns the *Patrol Name* attribute to the *Event* entity.

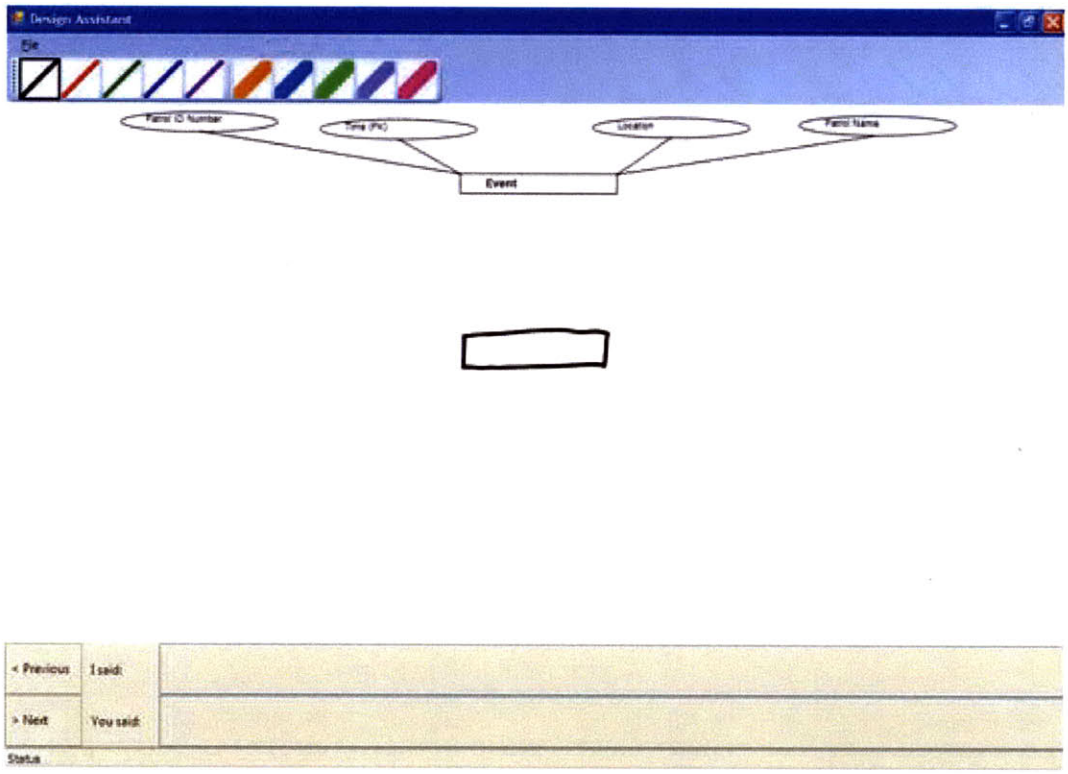


Figure 5-46: The user draws a new entity.

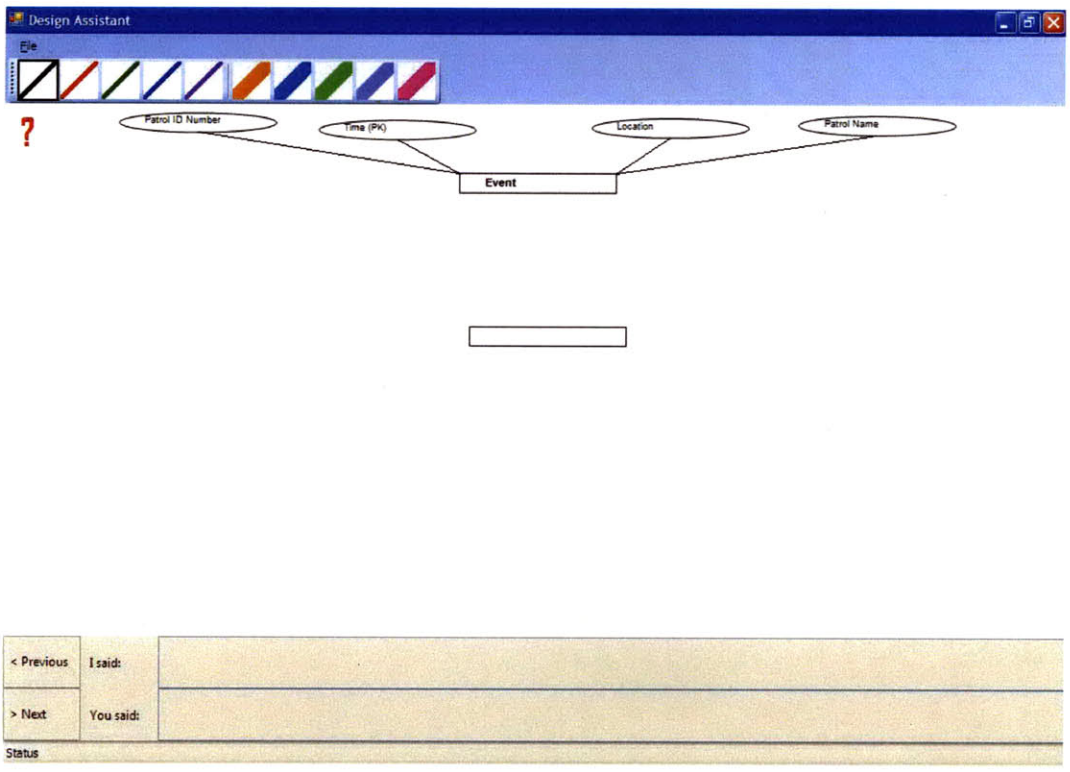


Figure 5-47: The system recognizes the user's input and adds a new entity to the entity-relationship diagram.

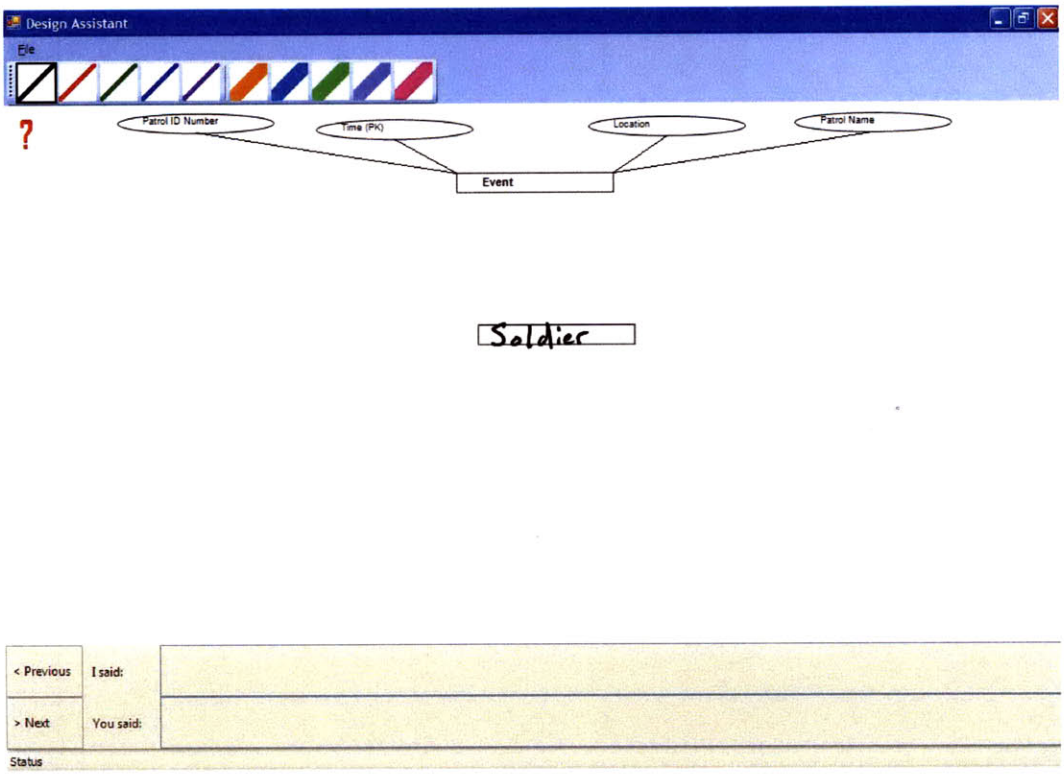


Figure 5-48: The user writes in the name for the new entity.

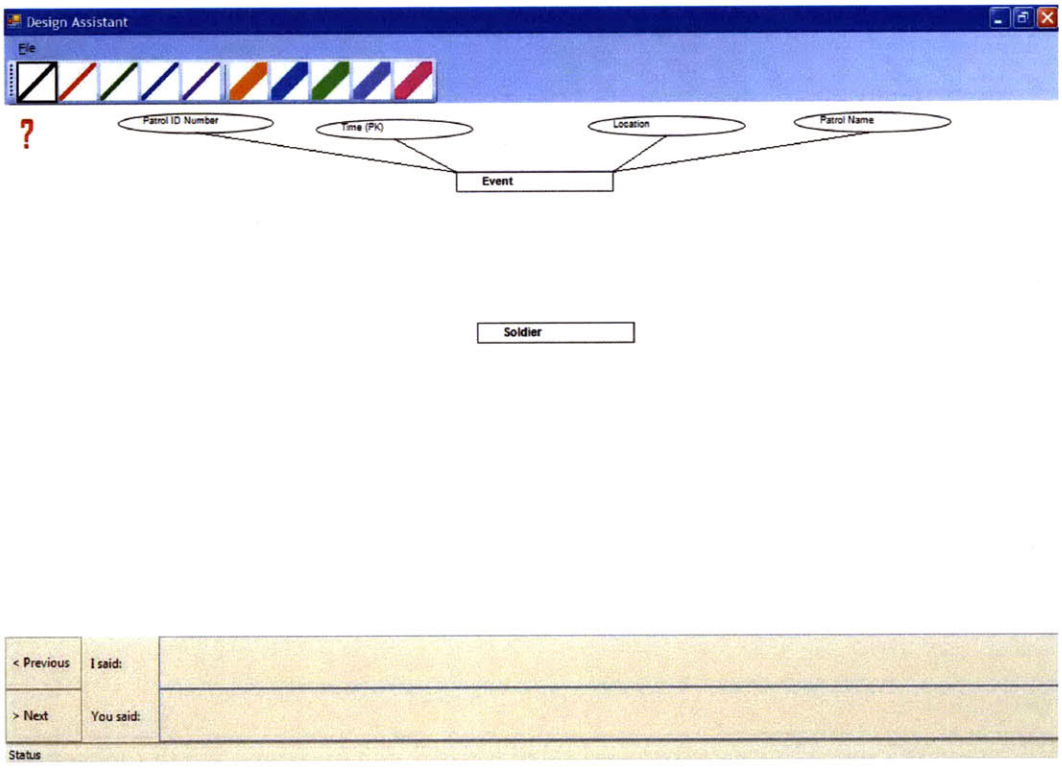


Figure 5-49: The system recognizes the user's handwriting and renames the new entity.

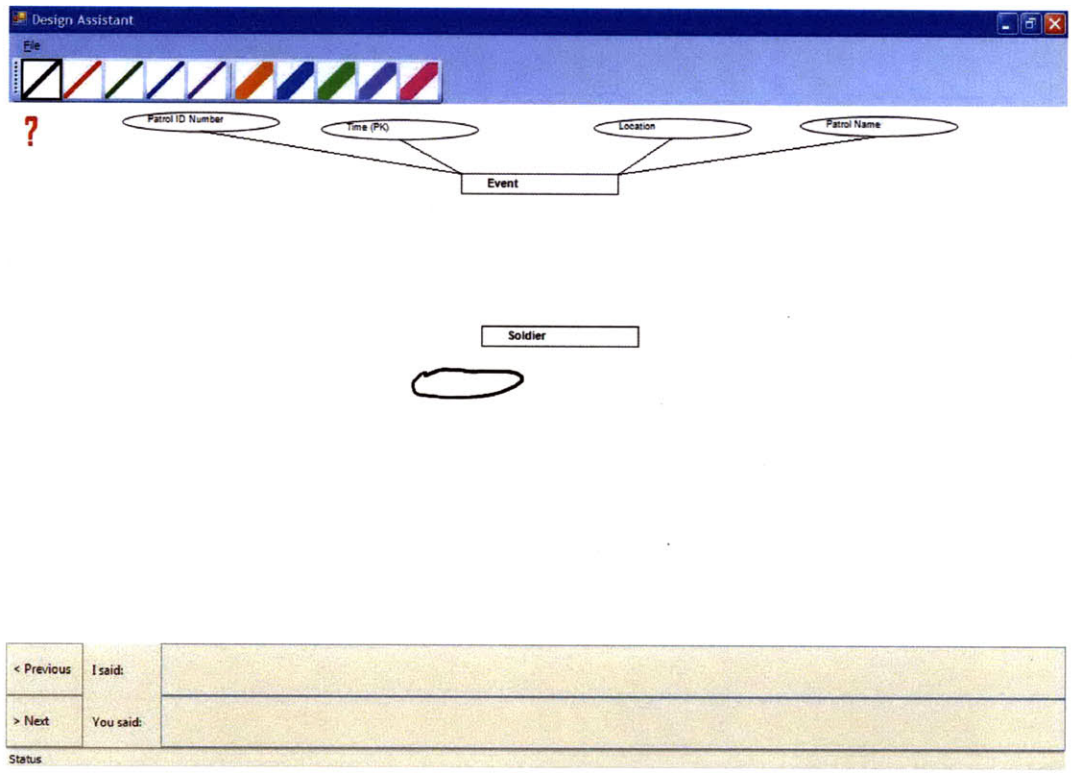


Figure 5-50: The user draws a new attribute.

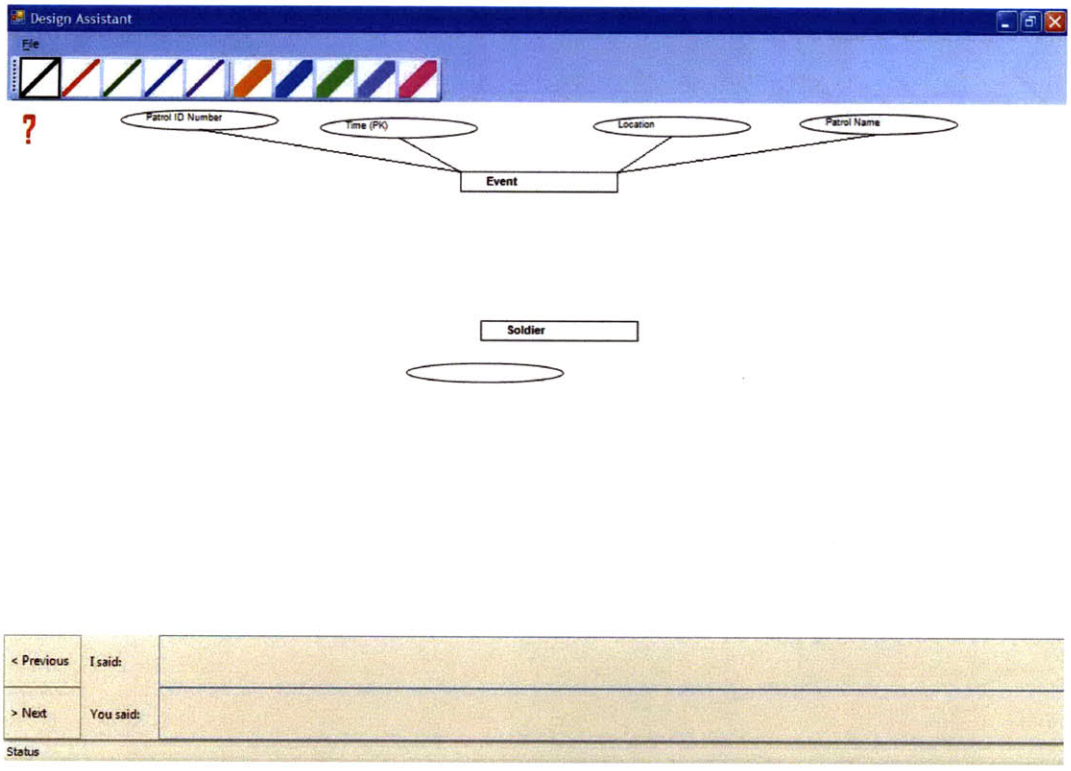


Figure 5-51: The system recognizes the user's input and adds a new attribute to the entity-relationship diagram.

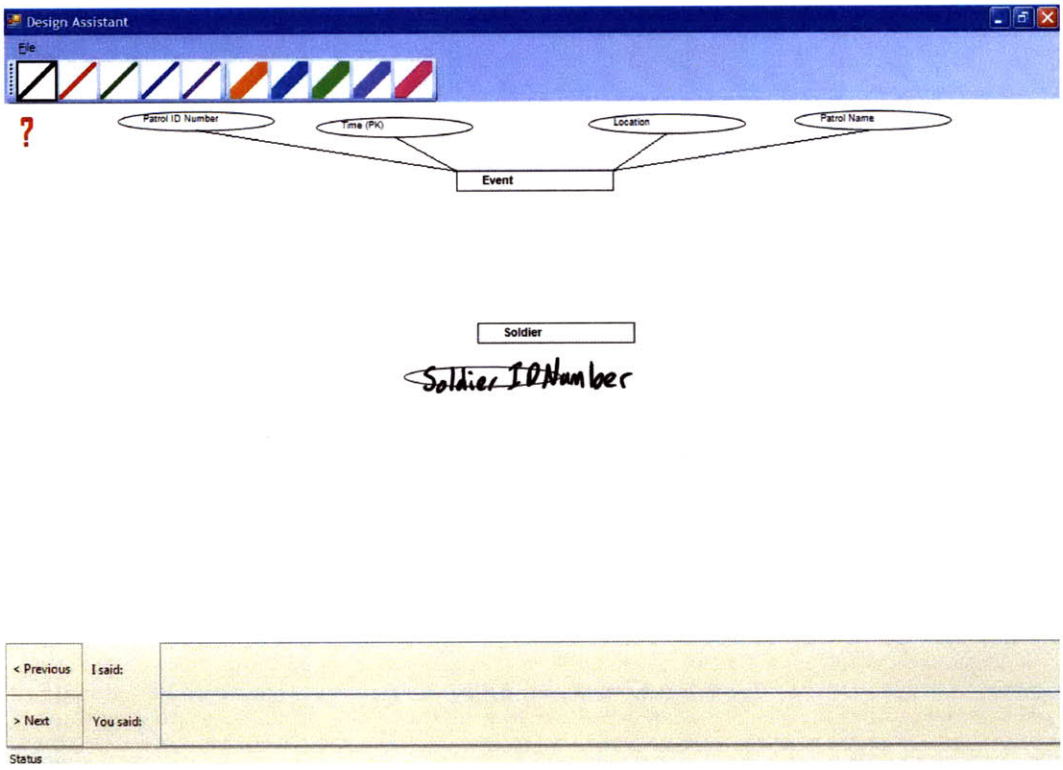


Figure 5-52: The user writes in the name for the new attribute.

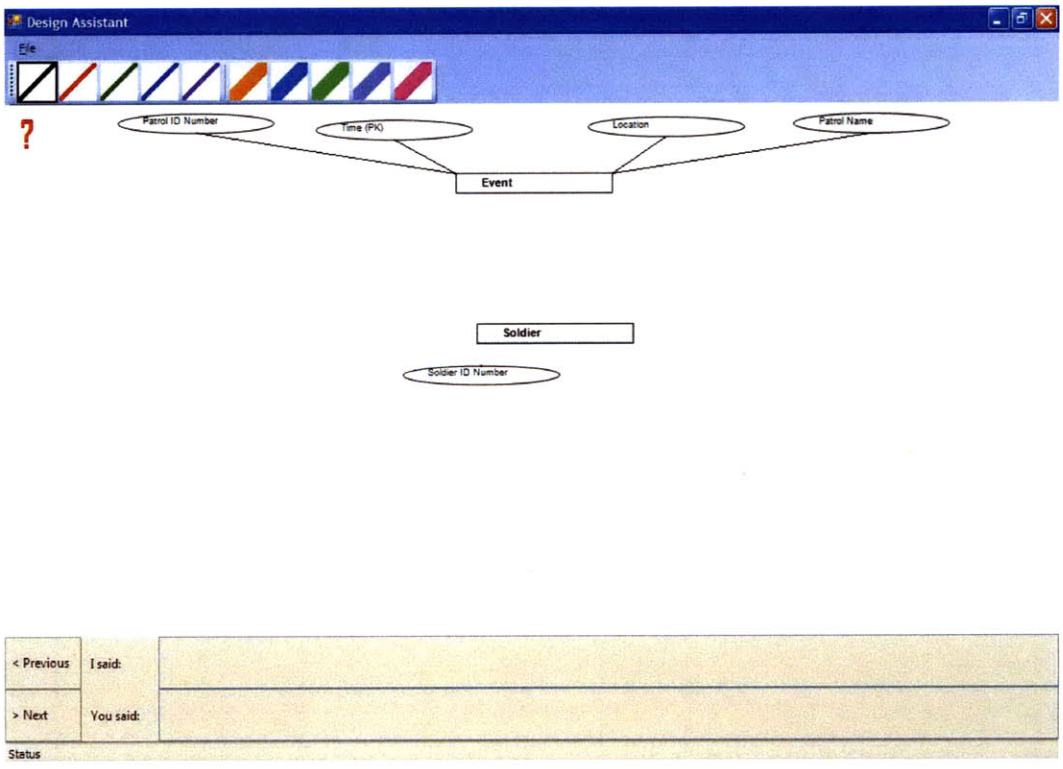


Figure 5-53: The system recognizes the user's handwriting and renames the attribute.

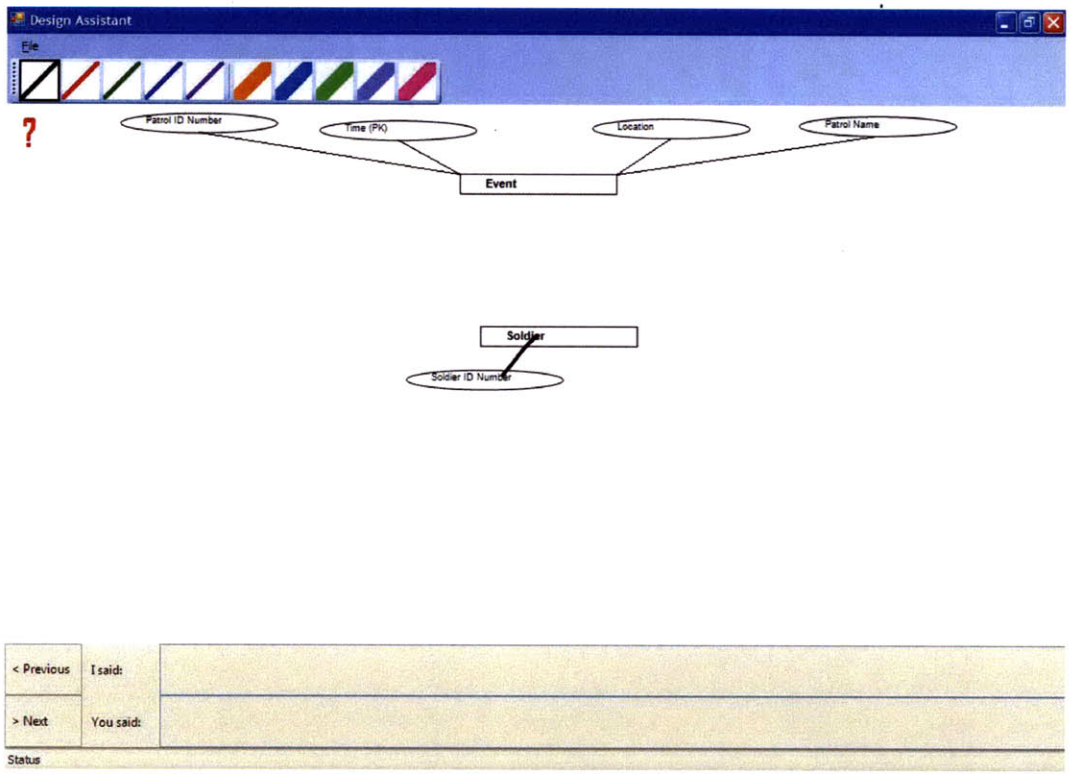


Figure 5-54: The user assigns the *Soldier ID Number* attribute to the *Soldier* entity.

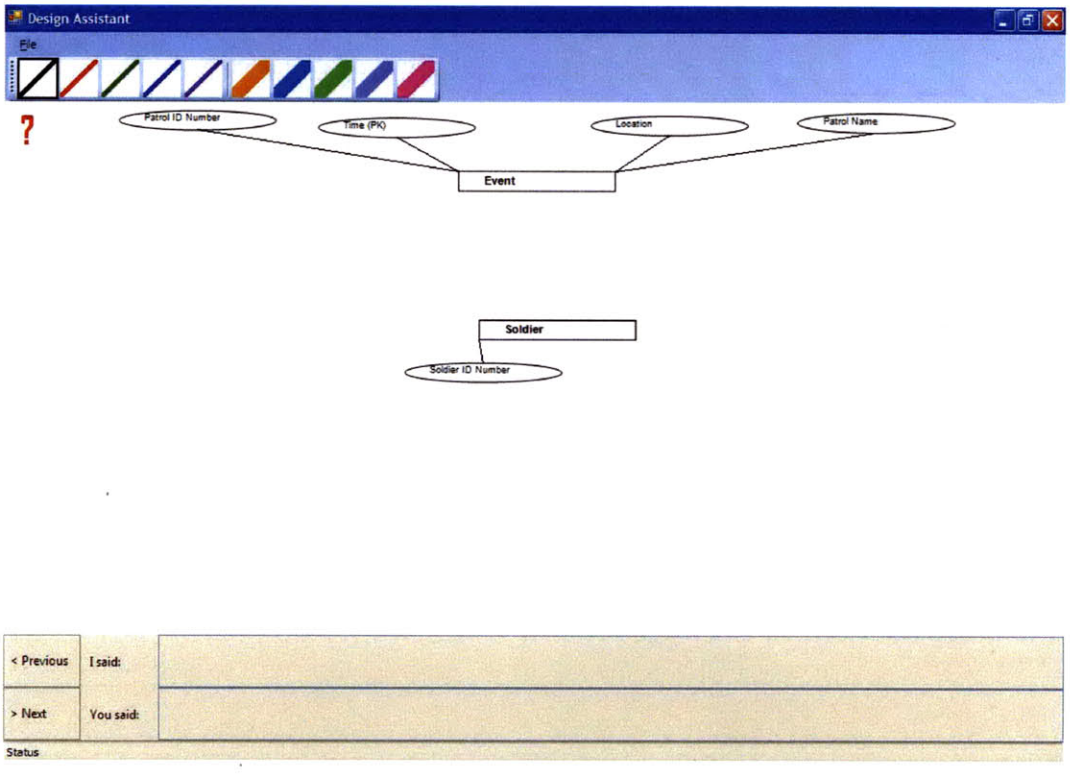


Figure 5-55: The system recognizes the user's input and assigns the *Soldier ID Number* attribute to the *Soldier* entity.

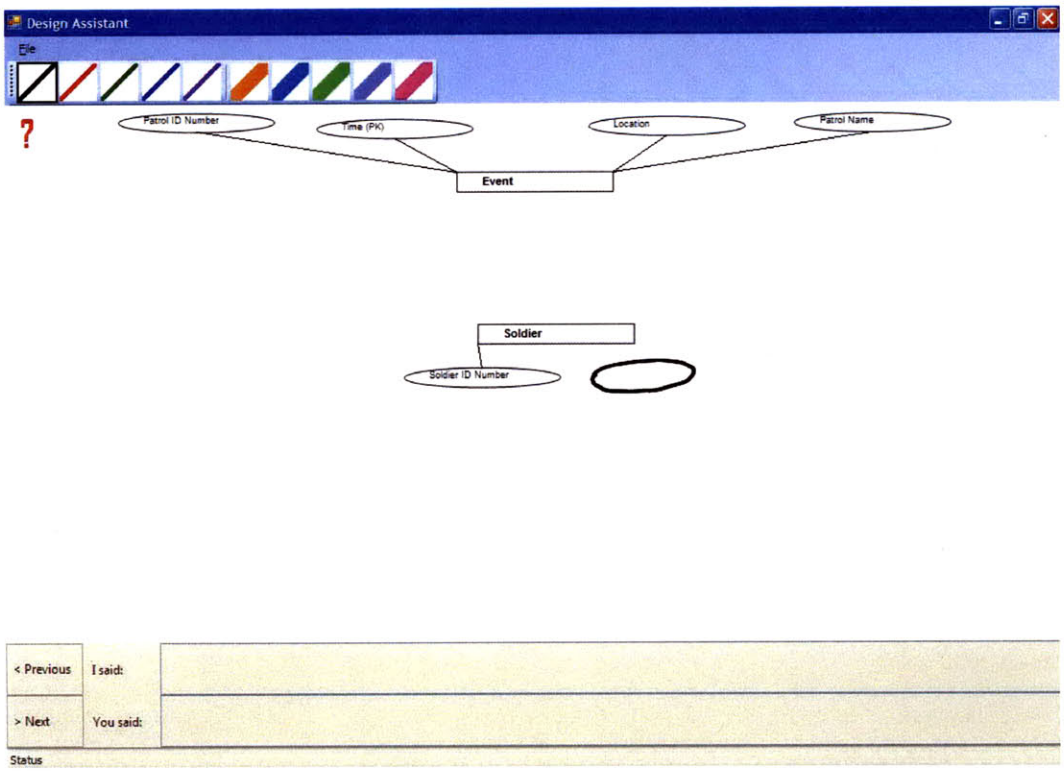


Figure 5-56: The user draws a new attribute.

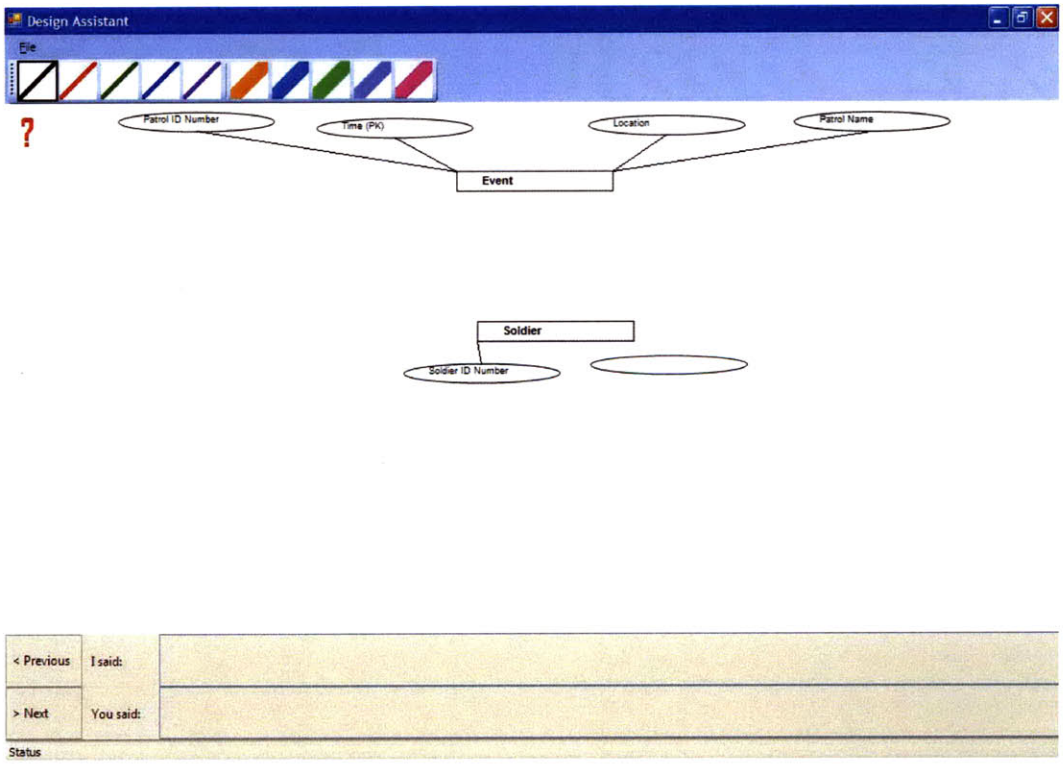


Figure 5-57: The system recognizes the user's input and adds a new attribute to the entity-relationship diagram.

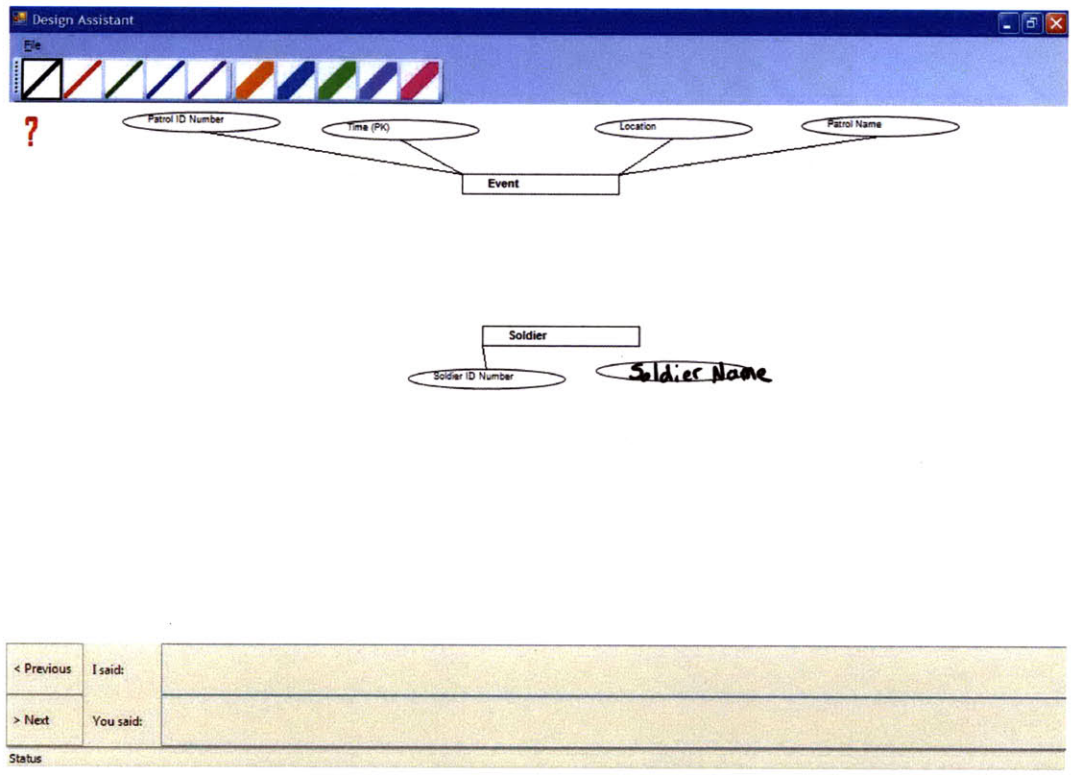


Figure 5-58: The user writes in a name for the new attribute.

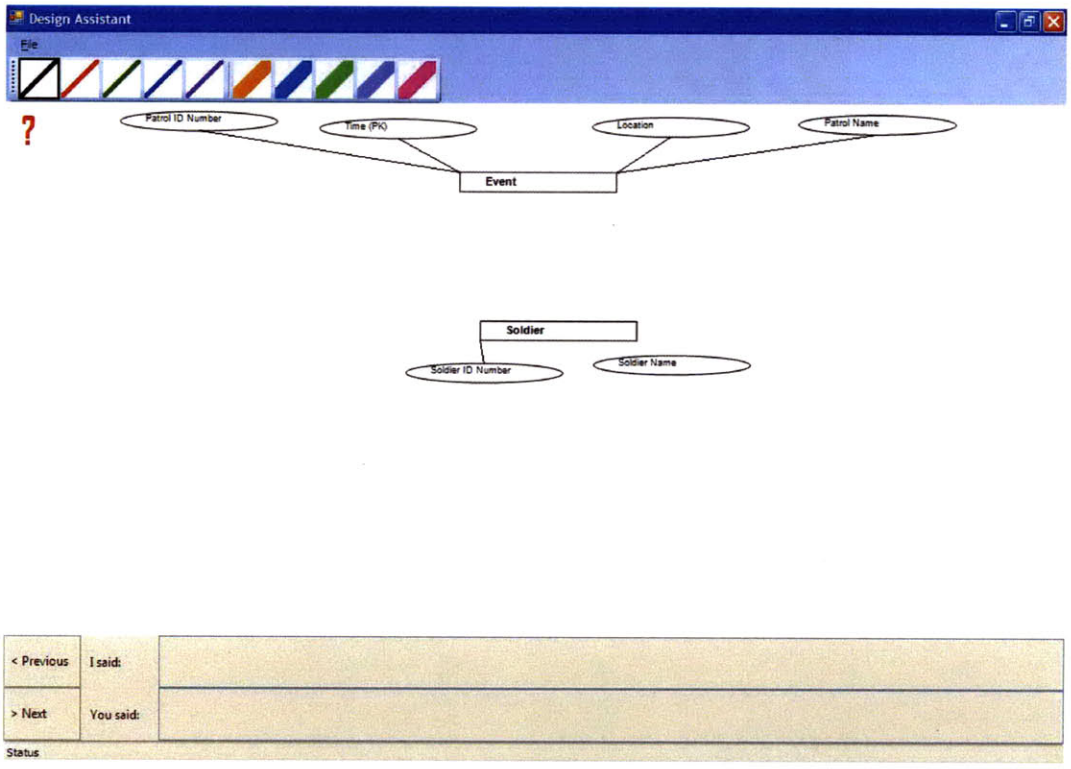


Figure 5-59: The system recognizes the user's handwriting and renames the attribute.

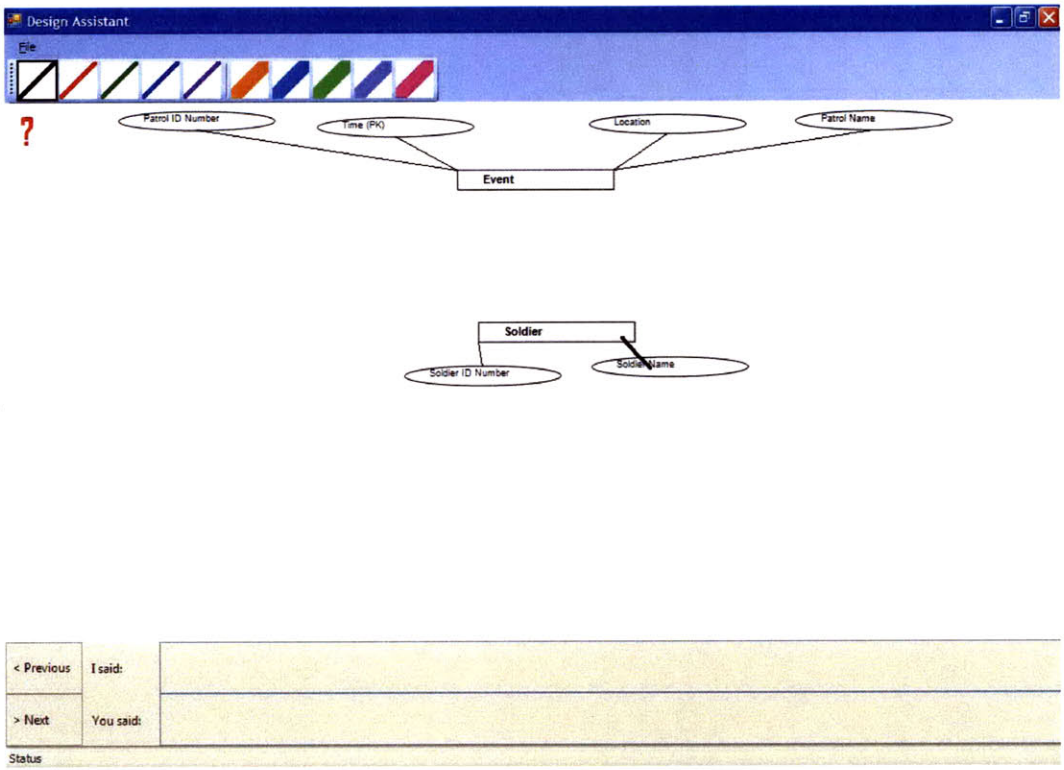


Figure 5-60: The user assigns the *Soldier Name* attribute to the *Soldier* entity.

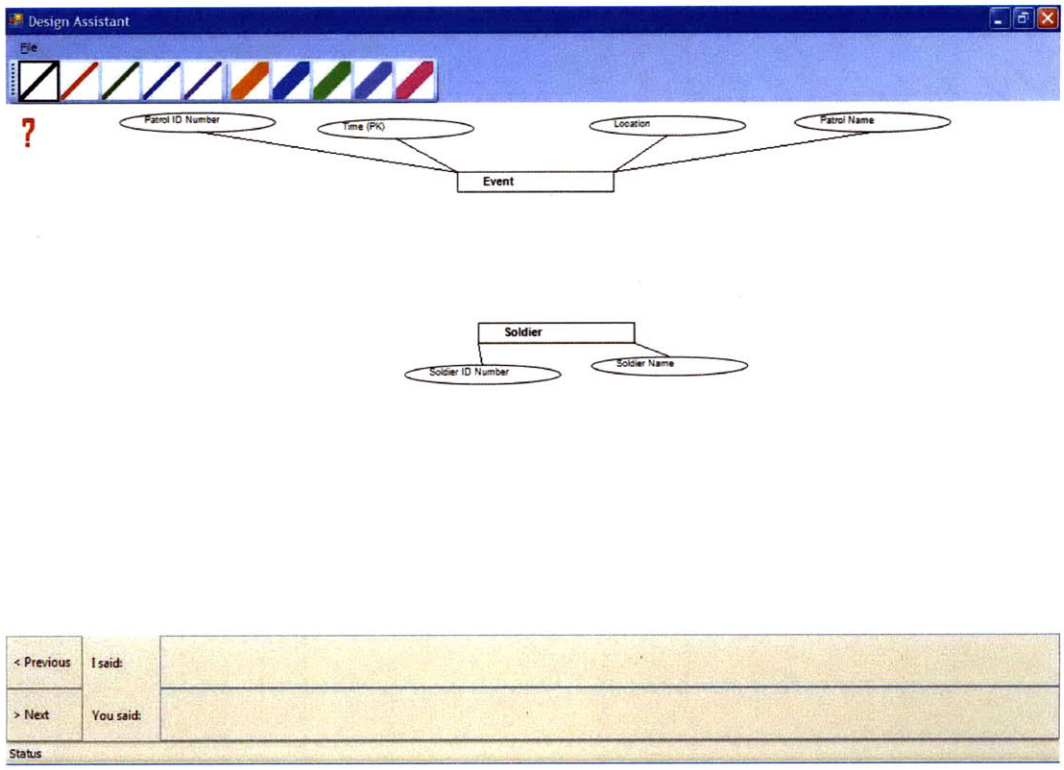


Figure 5-61: The system recognizes the user's input and assigns the *Solider Name* attribute to the *Soldier* entity.

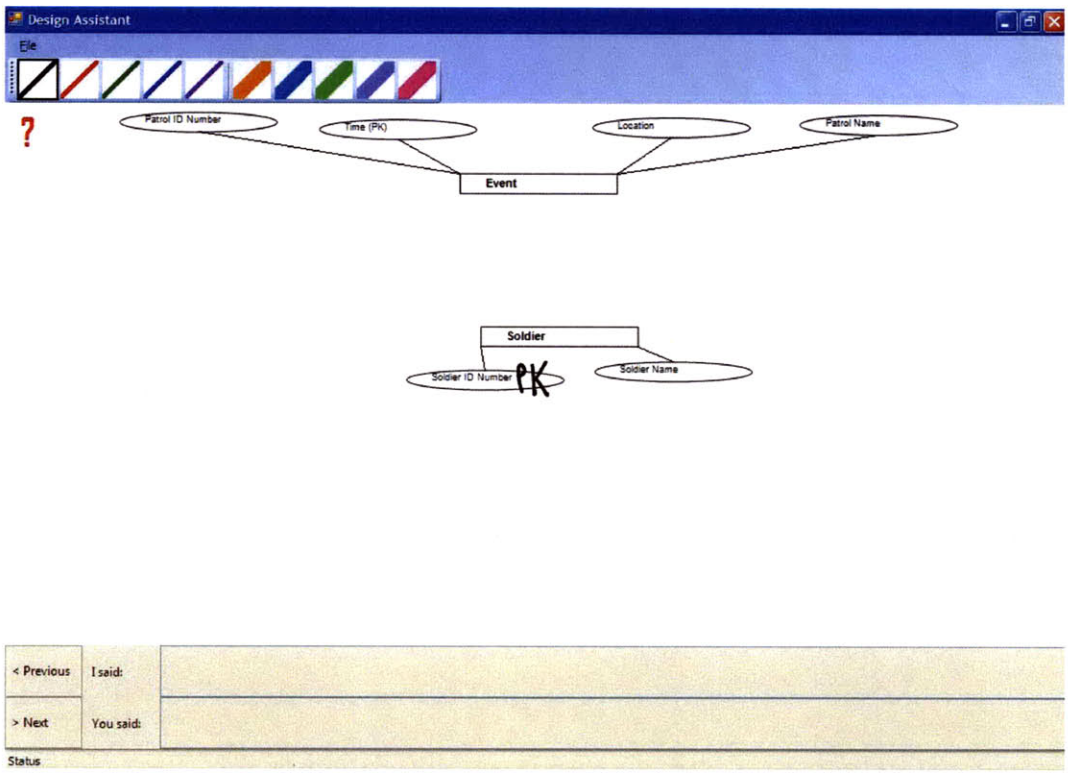


Figure 5-62: The user assigns the *Solider ID Number* attribute as the primary key for the *Soldier* entity.

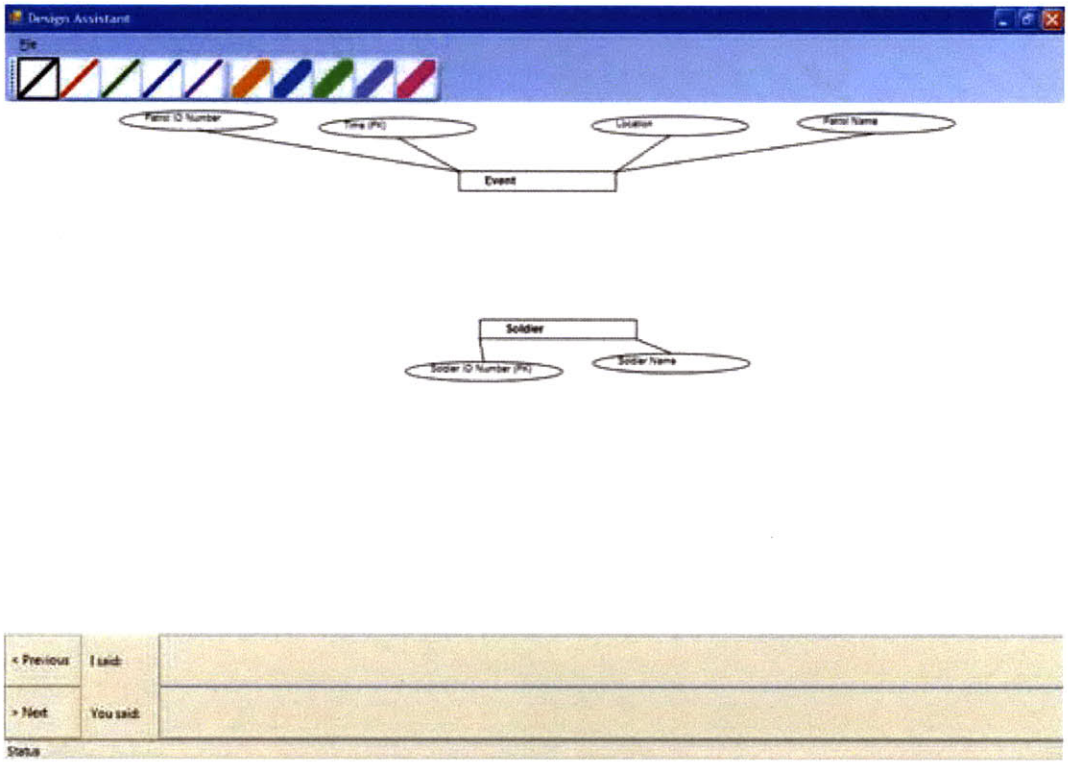


Figure 5-63: The system recognizes the user's input and assigns the *Solider ID Number* attribute as the primary key for the *Soldier* entity.

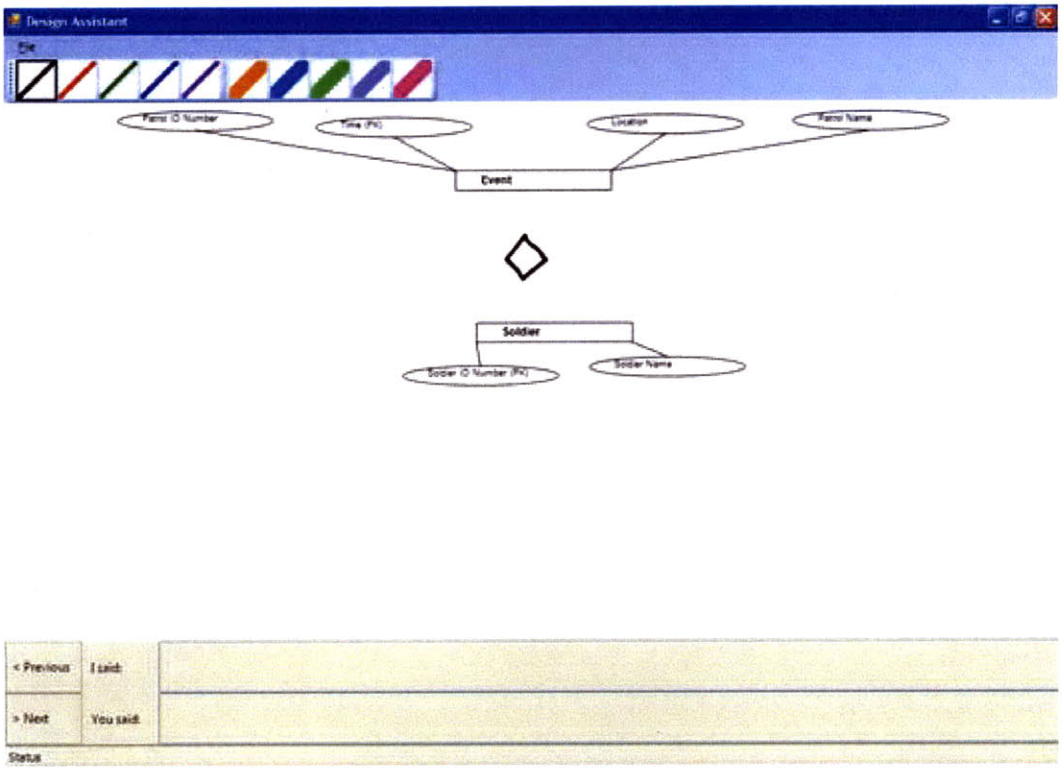


Figure 5-64: The user draws a new relation.

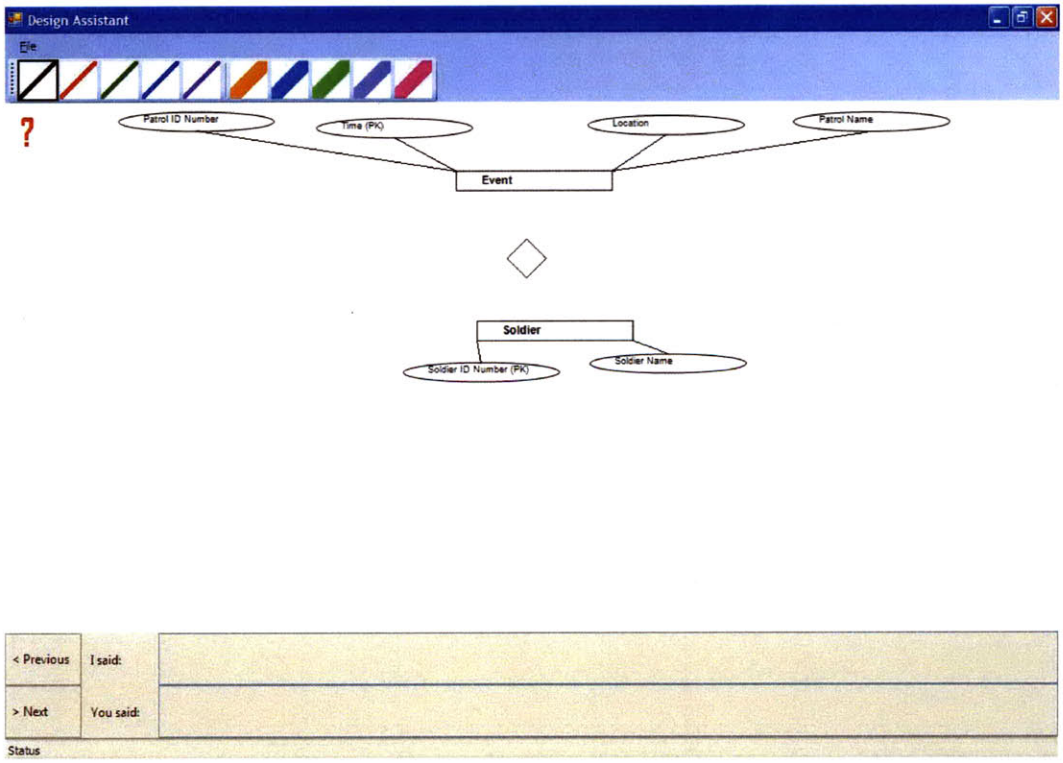


Figure 5-65: The system recognizes the user's input and adds a new relation to the entity-relationship diagram.

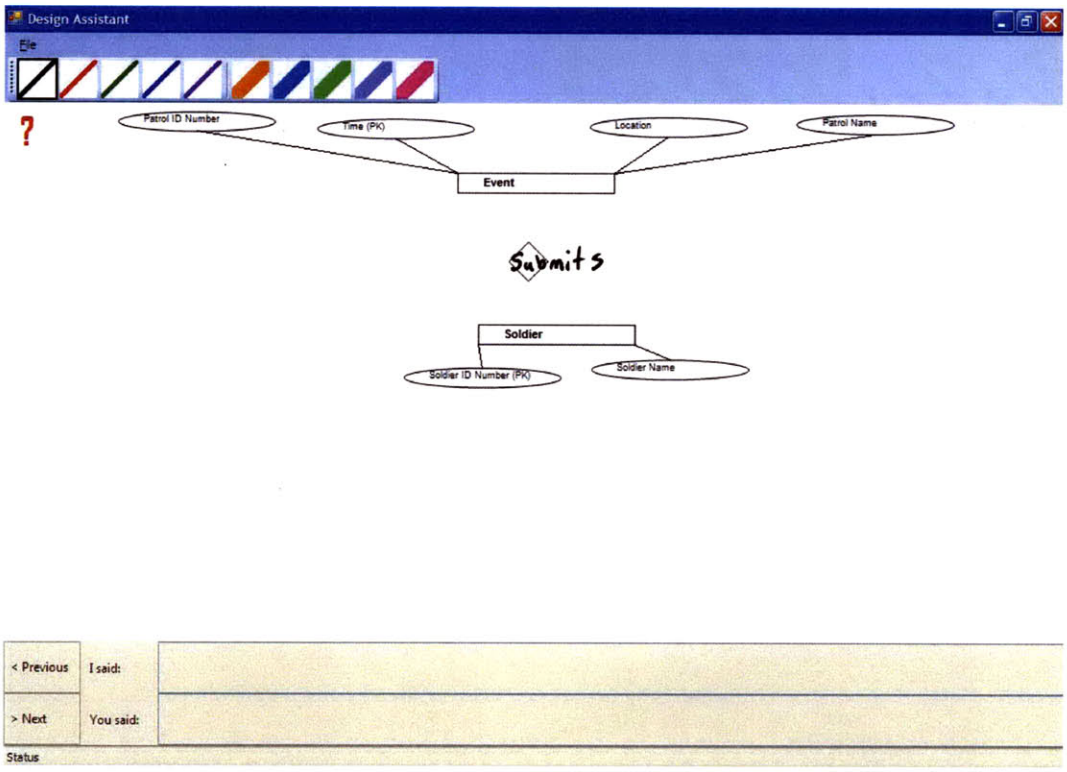


Figure 5-66: The user writes in a name for the new relation.

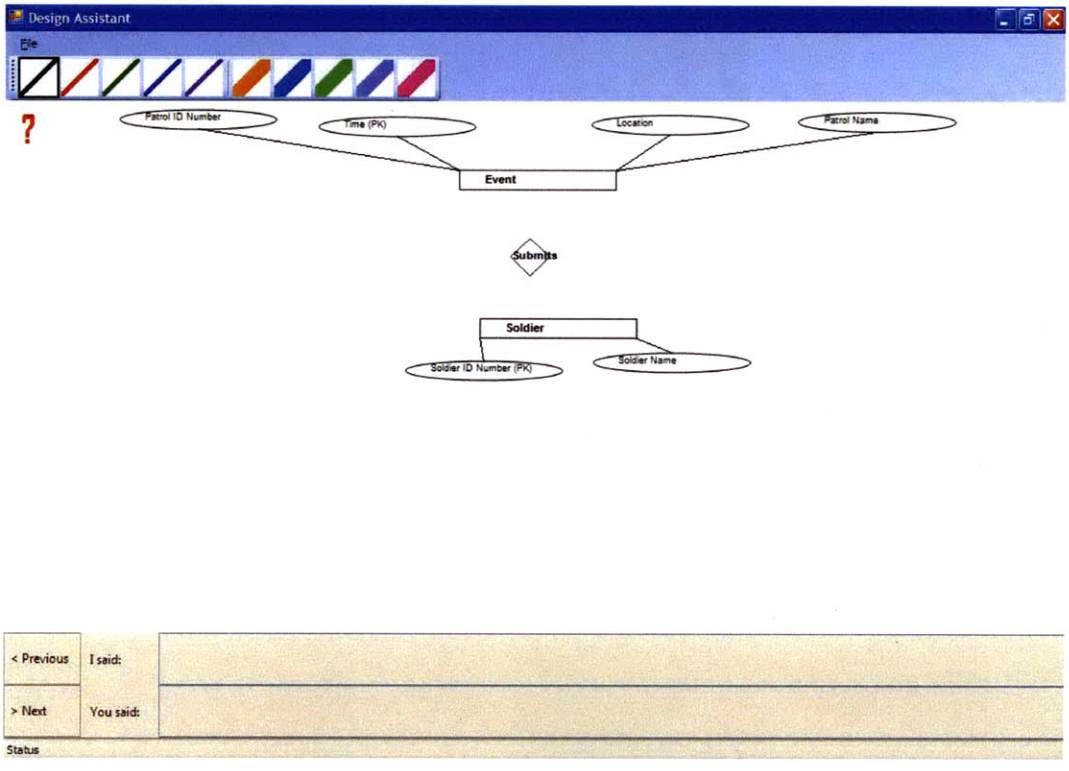


Figure 5-67: The system recognizes the user's handwriting and renames the relation.

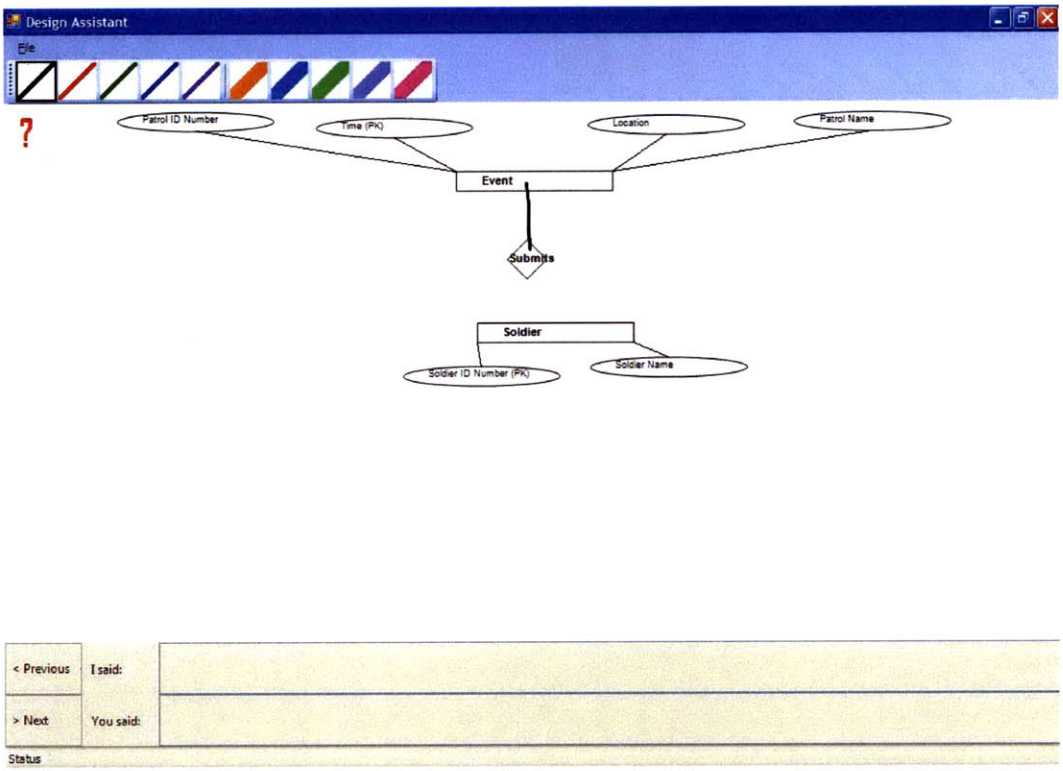


Figure 5-68: The user assigns the *Event* entity to the *Submits* relation.

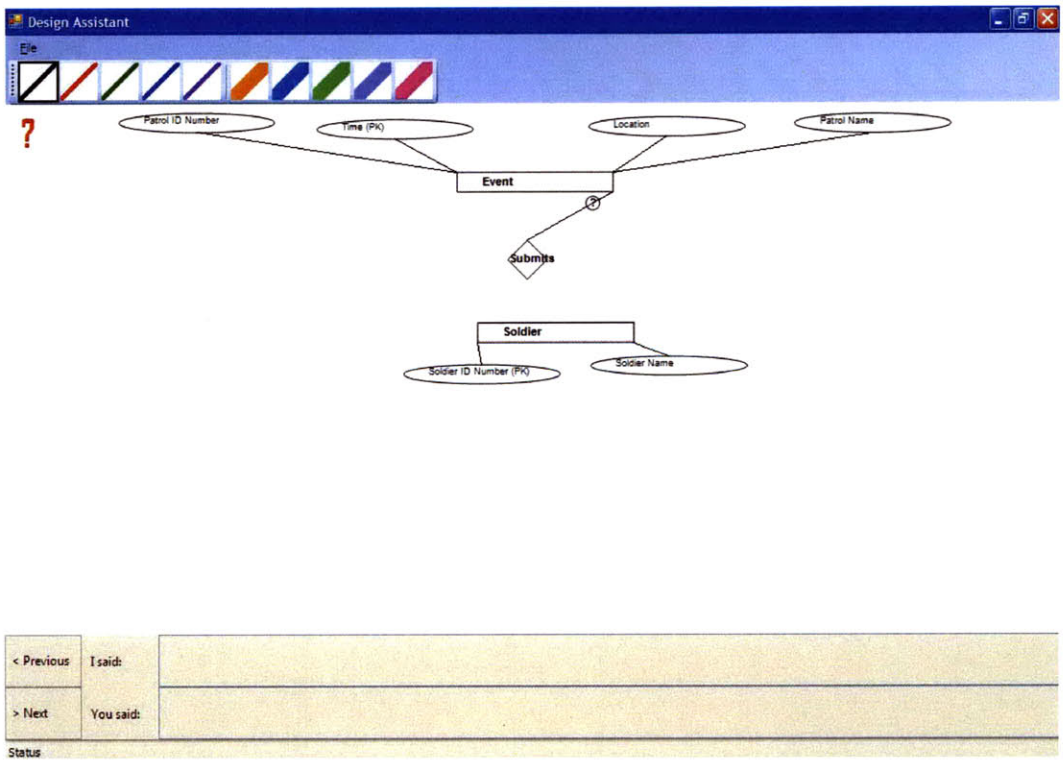


Figure 5-69: The system recognizes the user's input and assigns the *Event* entity to the *Submits* relation.

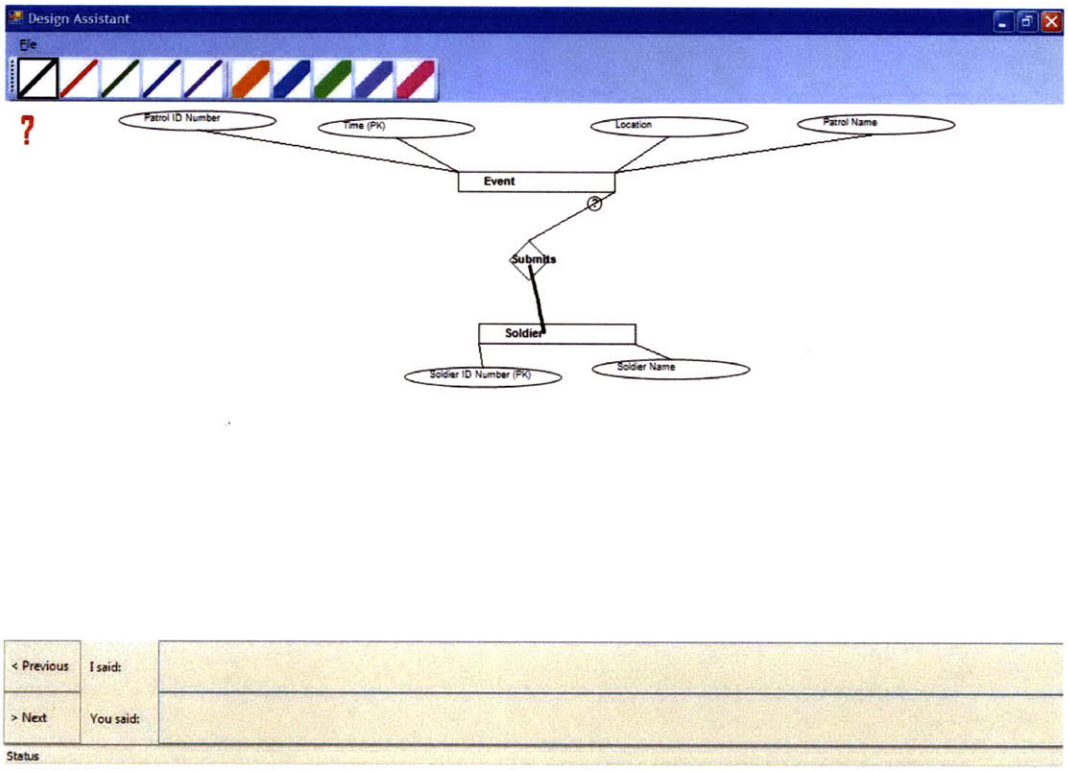


Figure 5-70: The user assigns the *Soldier* entity to the *Submits* relation.

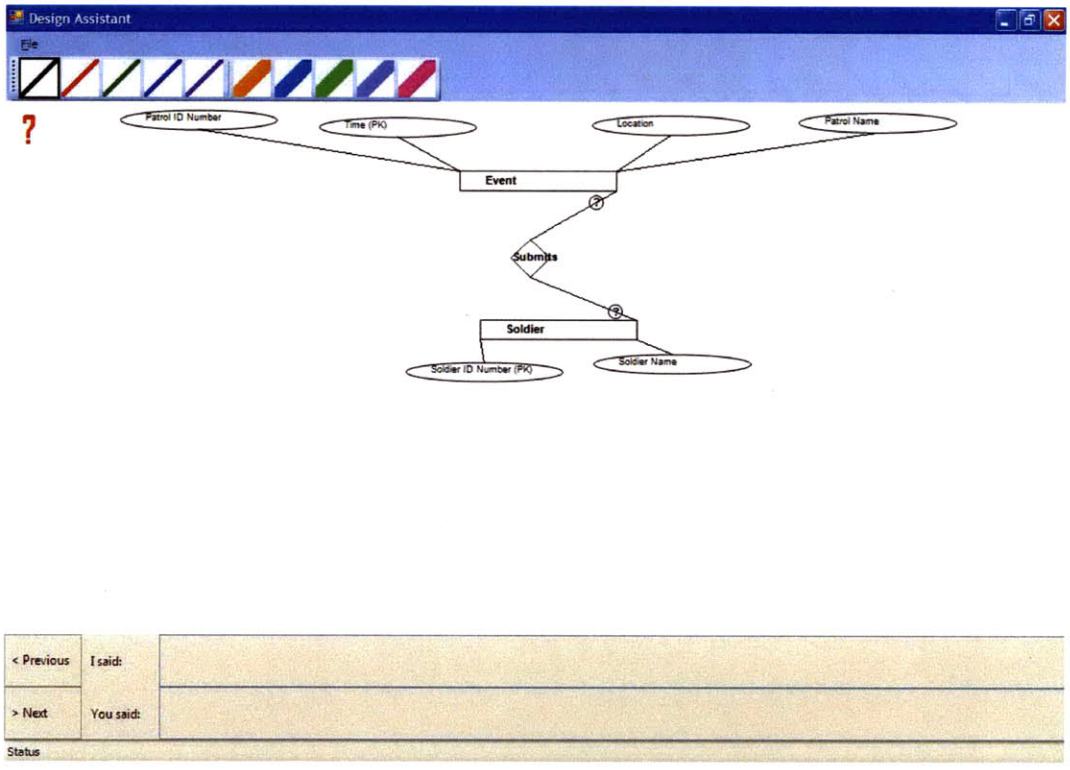
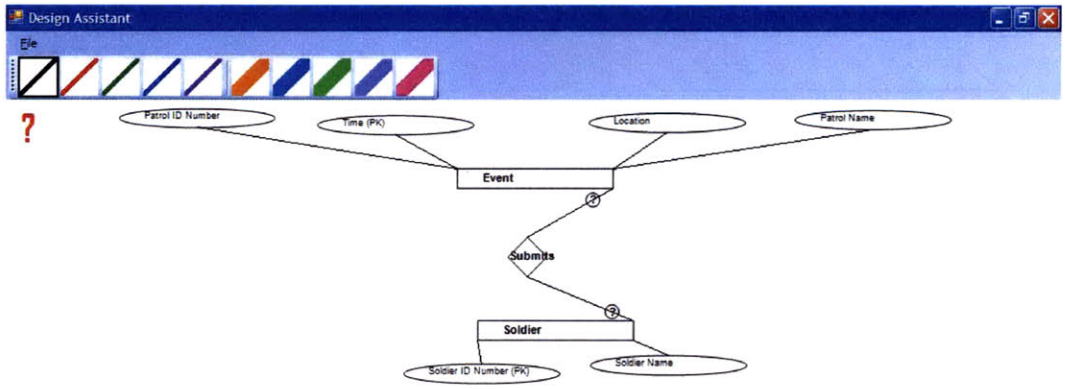


Figure 5-71: The system recognizes the user's input and assigns the *Soldier* entity to the *Submits* relation.



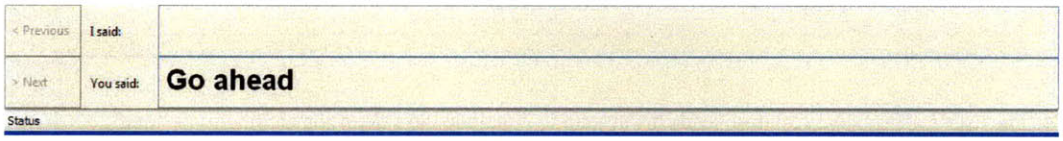
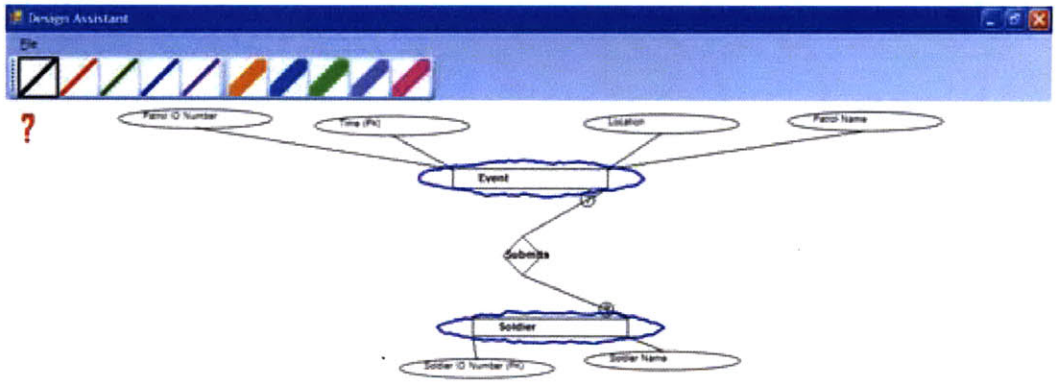


Figure 5-72: The users tells the system to start asking questions.



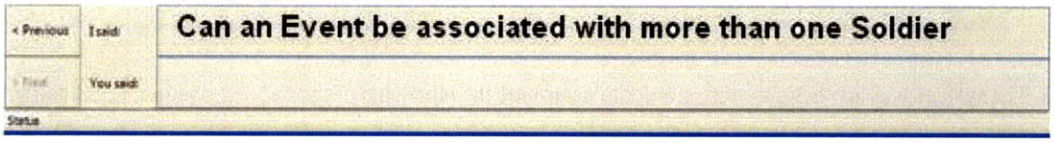


Figure 5-73: The system asks a question in order to determine the first side of the *Submits* relation's cardinality.

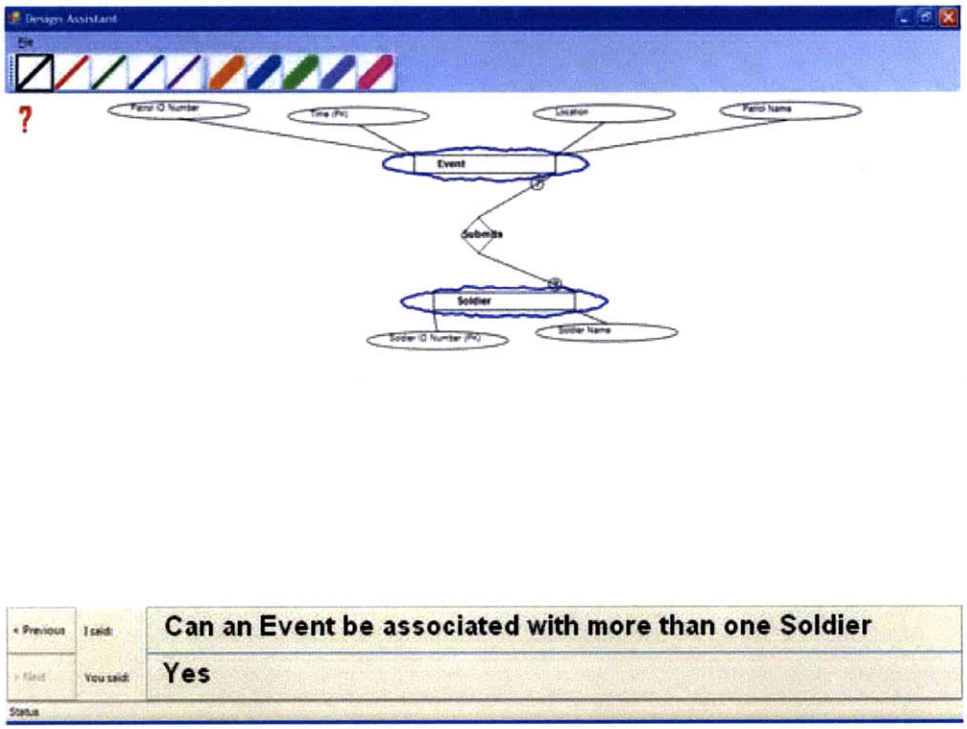


Figure 5-74: The user's affirmative response leads the system to conclude the side of the cardinality is "one".

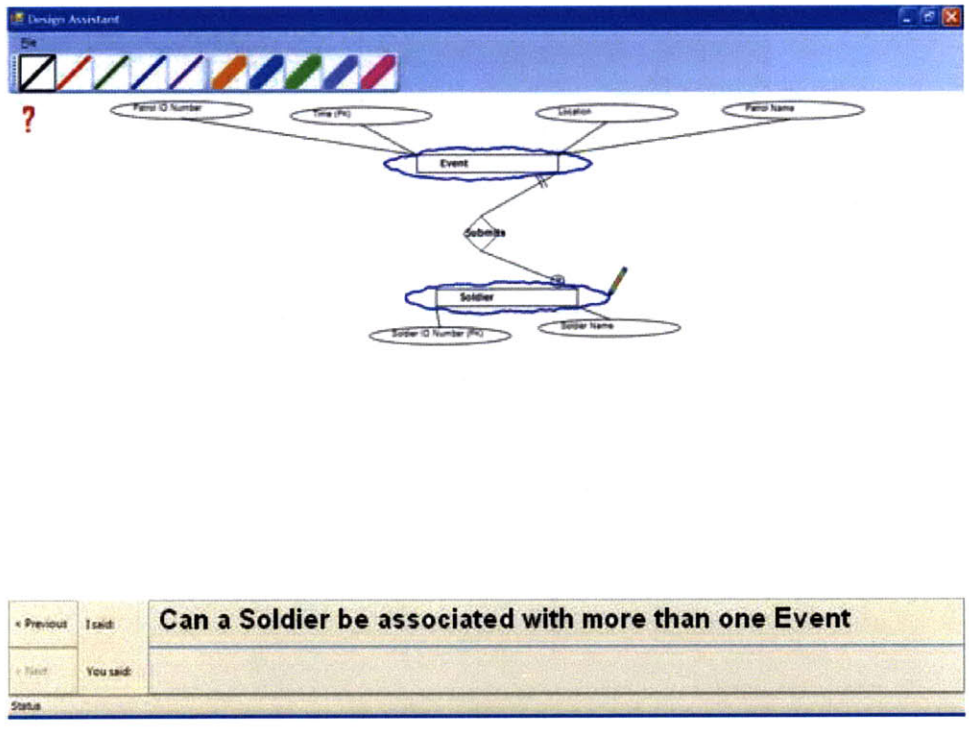


Figure 5-75: The system asks another question in order to determine the other side of the *Submits* relation's cardinality.

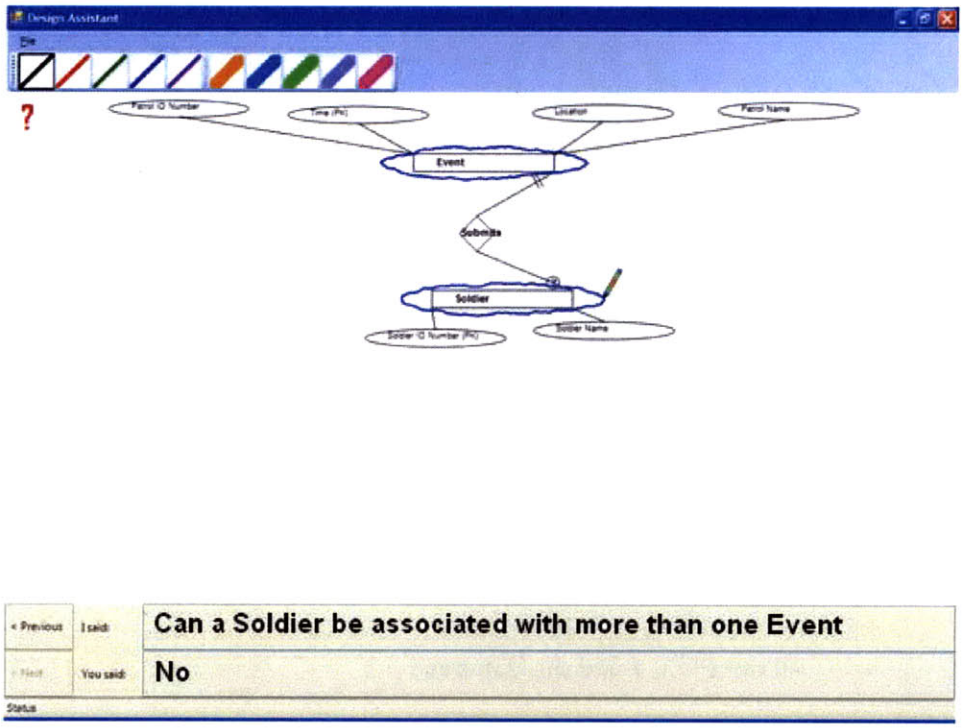


Figure 5-76: The user's negative response leads the system to conclude the side of the cardinality is "many".

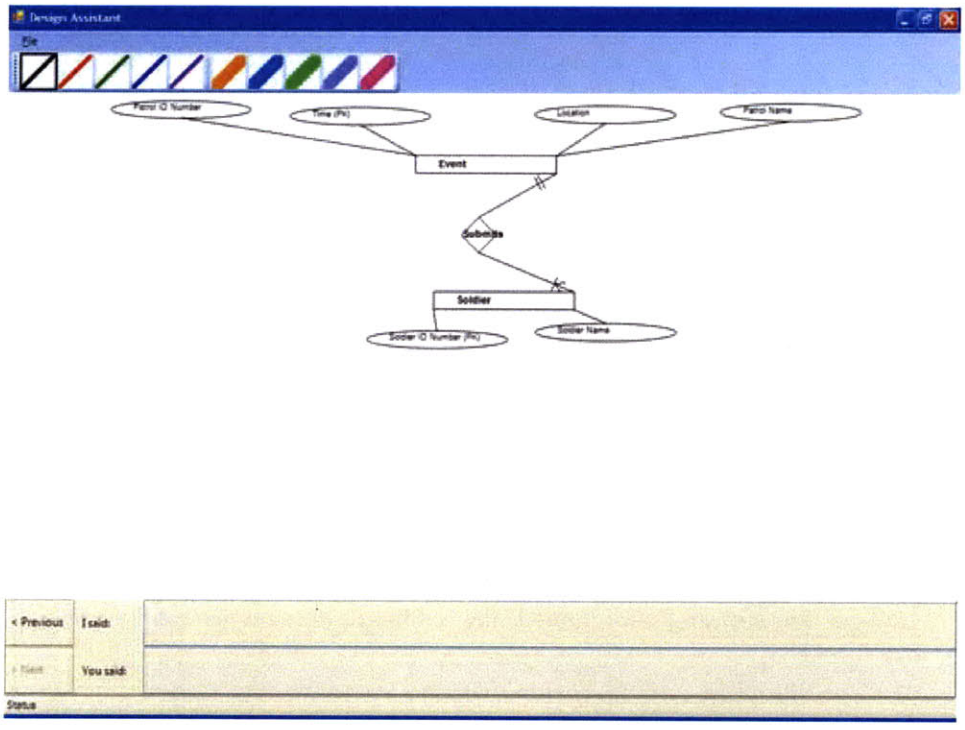


Figure 5-77: Based on the user's responses to the last two questions, the system determines that the *Submits* relation to have a "one-to-many" cardinality.

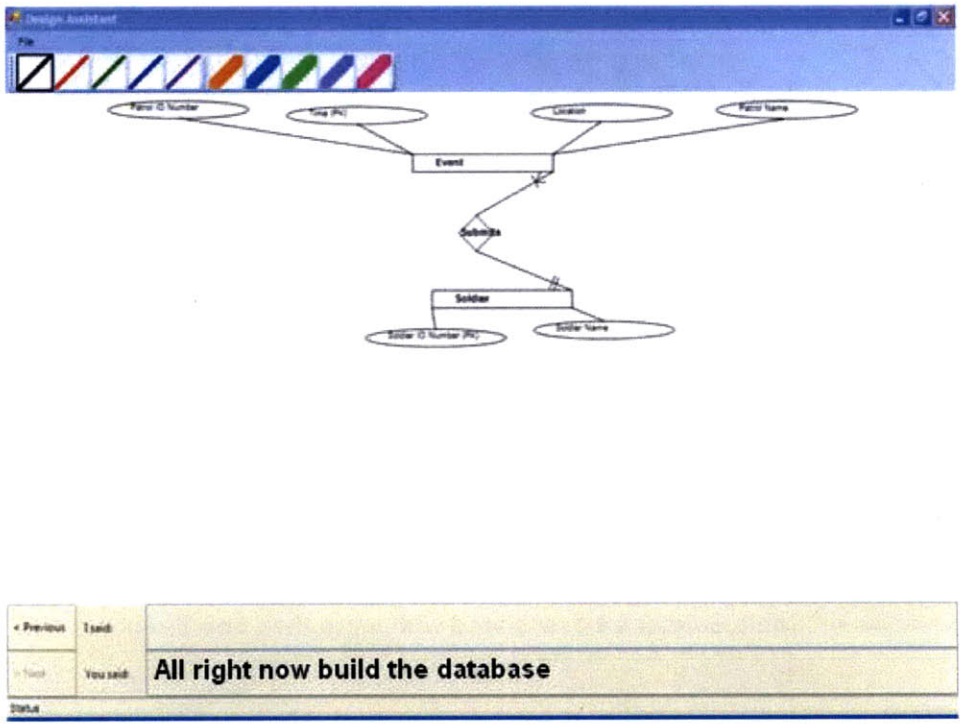


Figure 5-78: The user is satisfied with the design and commands the system to build and run the relational database.

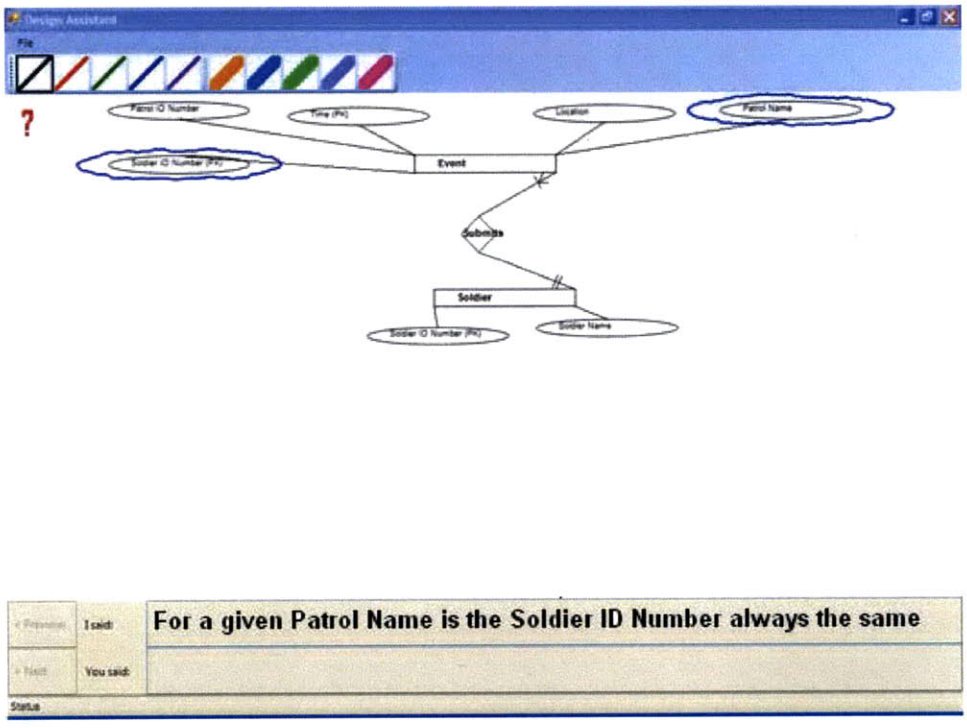
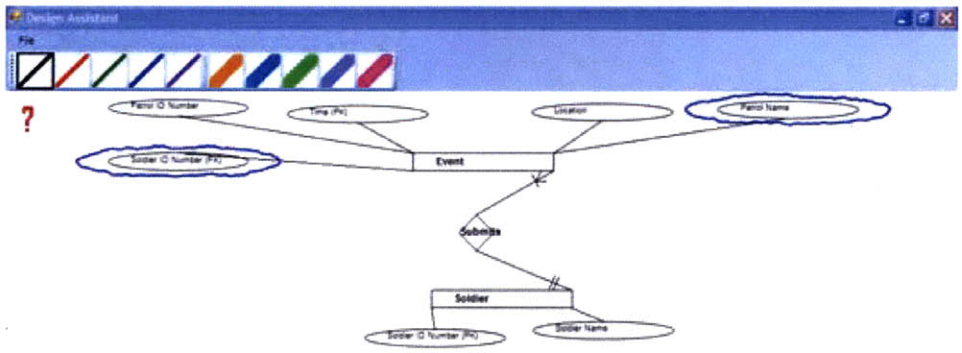


Figure 5-79: After the database has been implemented and has some stored data, the system determines that there is a possible functional dependency, *Patrol Name* → *Soldier ID Number*, and wants to determine it this FD is a real FD for the database.



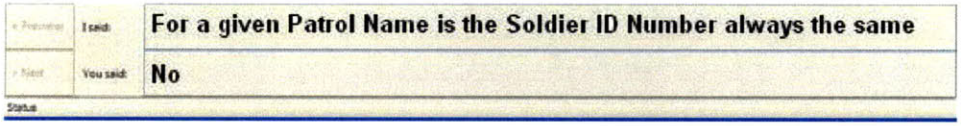
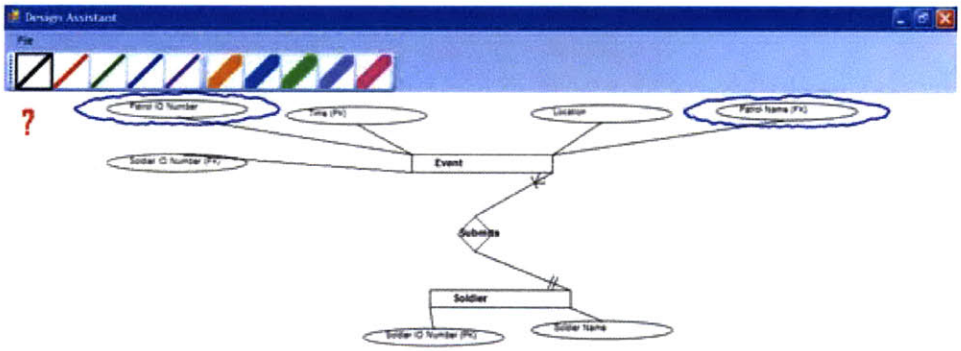


Figure 5-80: The user responds negatively, which means that the *Patrol Name* → *Soldier ID Number* FD candidate is not a real FD for the database.



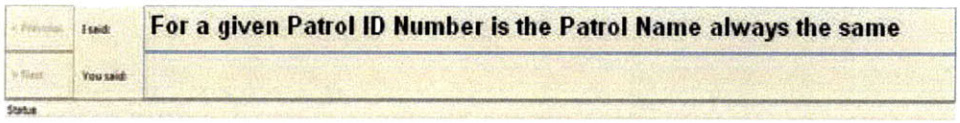


Figure 5-81: The system also determines that there is another possible functional dependency, *Patrol ID Number* → *Patrol Name*, and wants to determine it this FD is a real FD for the database.

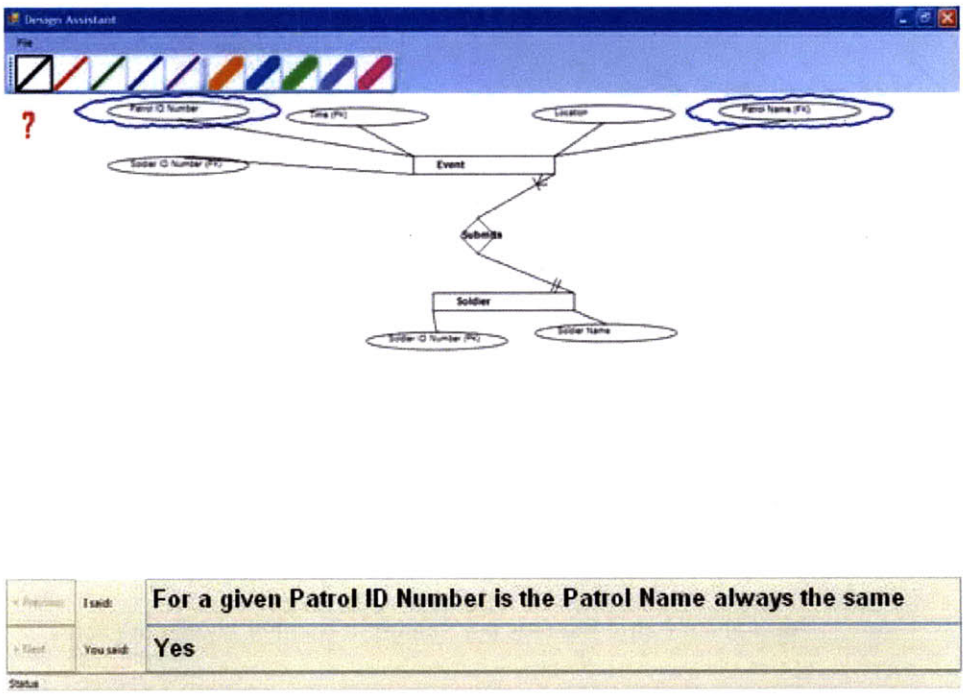


Figure 5-82: The user responds positively, which means that the *Patrol ID Number* → *Patrol Name* is a real FD for this database.

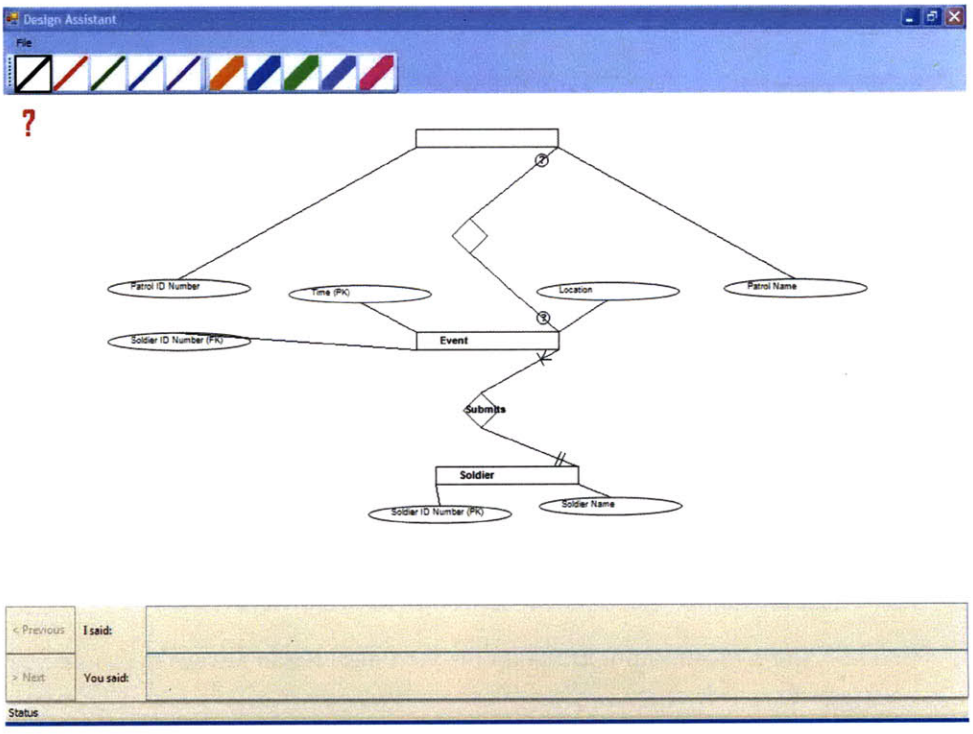


Figure 5-83: The new FD violates BCNF, so the system normalizes the database in order to address this new redundancy.

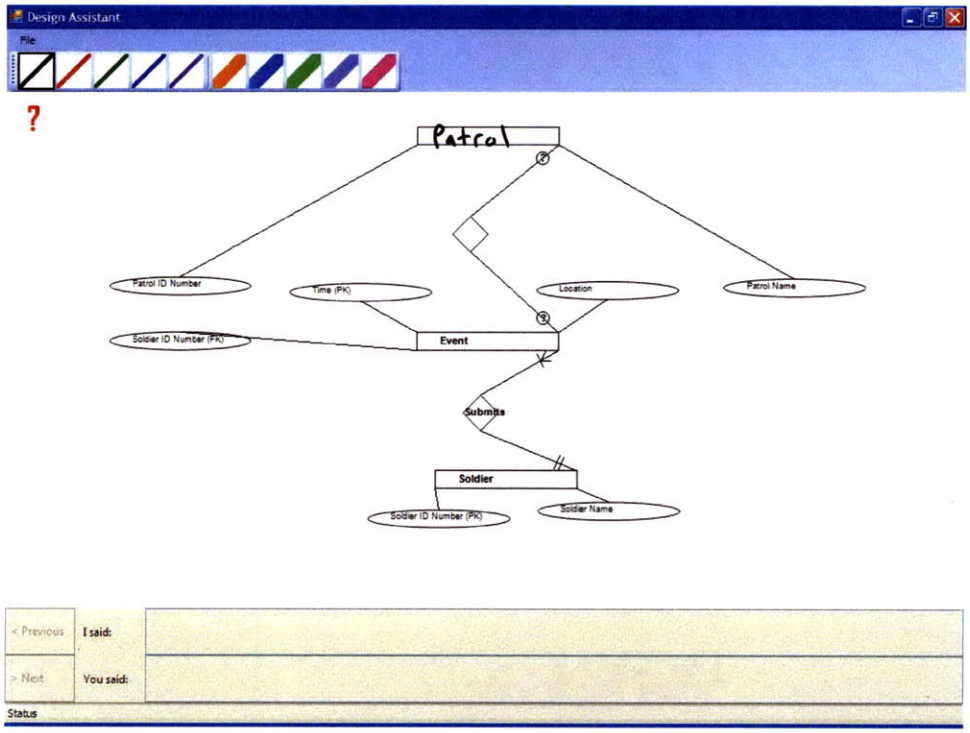


Figure 5-84: The user writes in the name for the new entity create d by the normalization process.

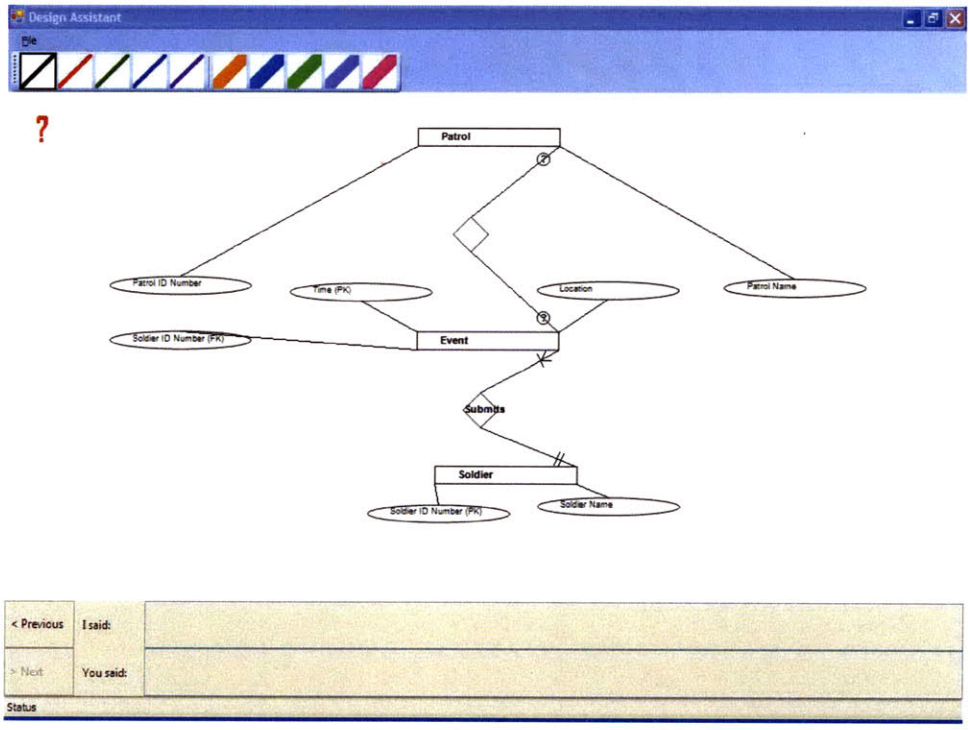


Figure 5-85: The system recognizes the user's handwriting and renames the entity.

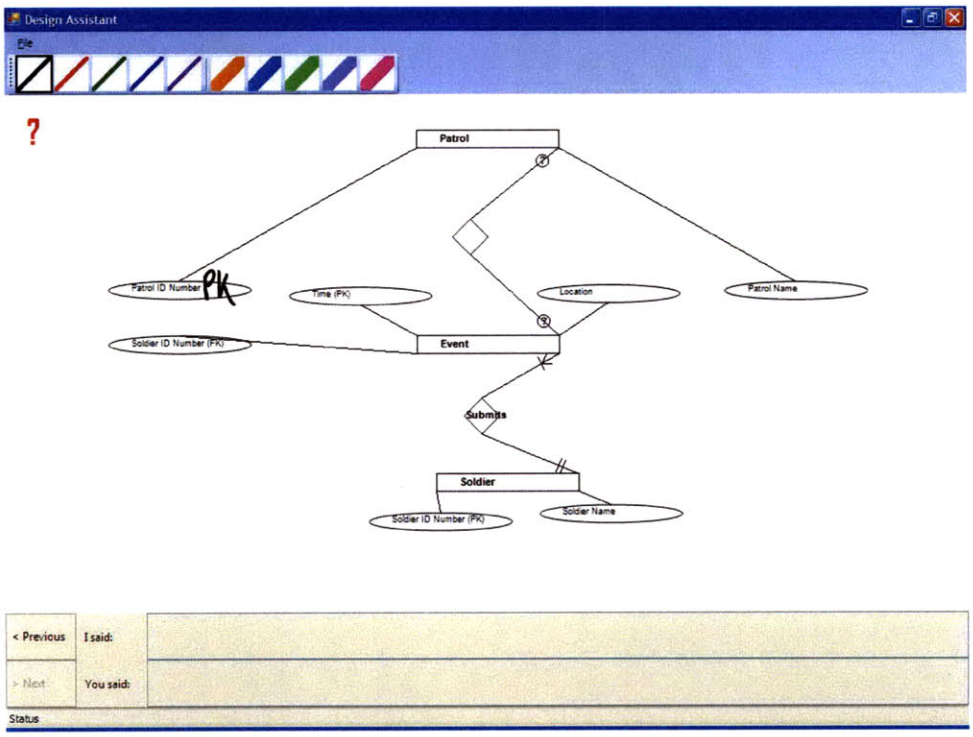


Figure 5-86: The user assigns the *Patrol ID Number* attribute to be the primary key for the *Patrol* entity.

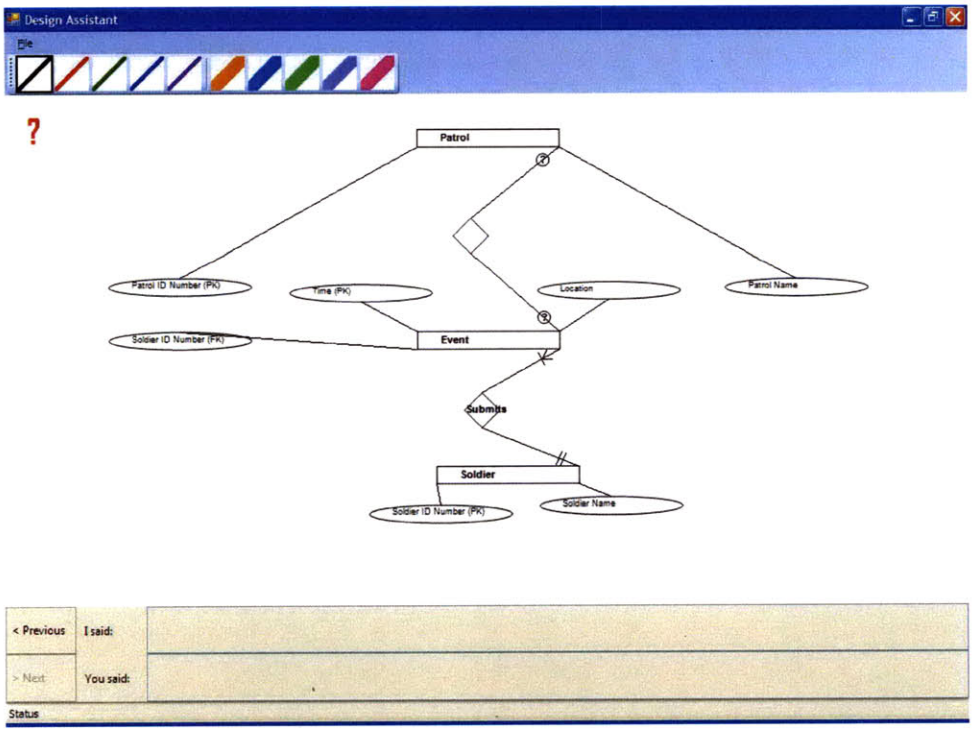


Figure 5-87: The system recognizes the user's input and assigns the *Patrol ID Number* to be the primary key for the *Patrol* entity.

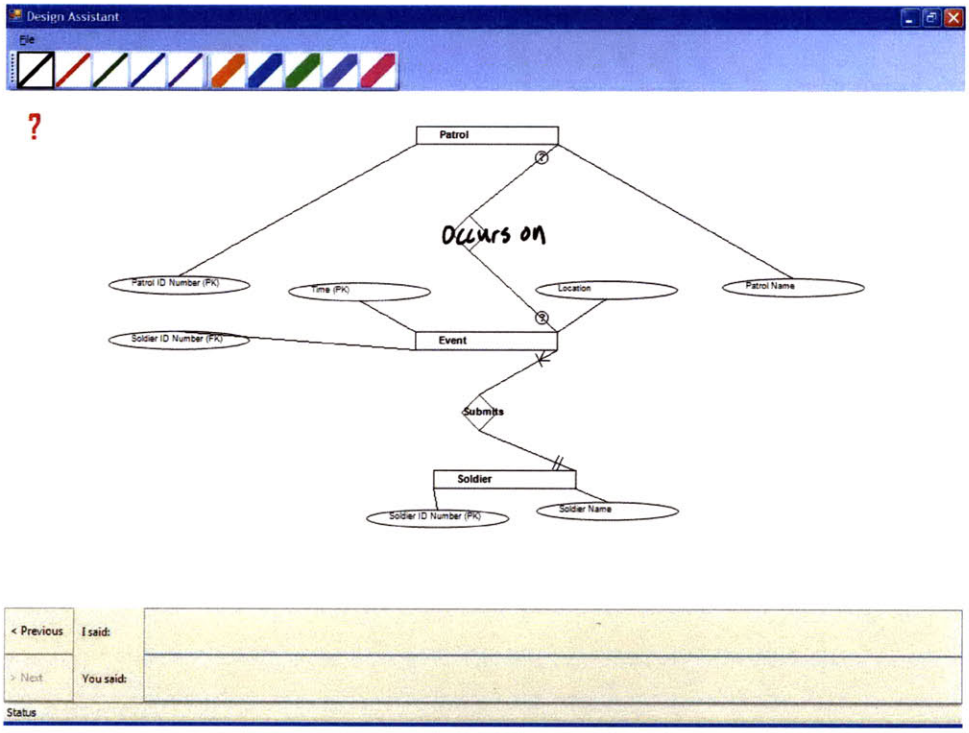


Figure 5-88: The user writes in the name for the new relation created by the normalization process.

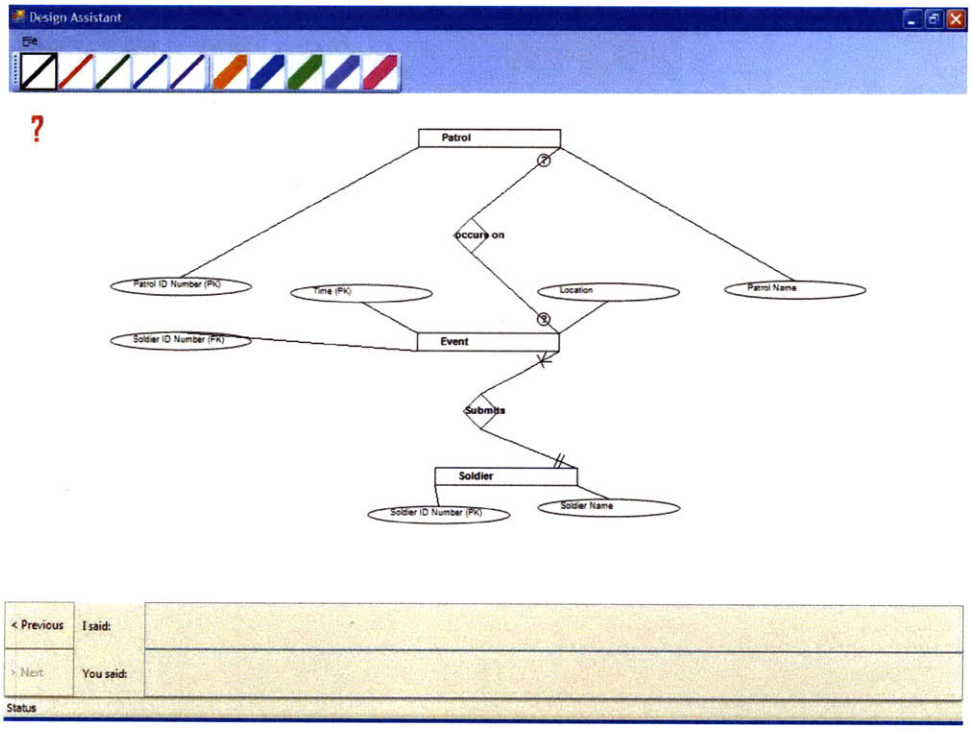


Figure 5-89: The system recognizes the user's handwriting and renames the relation.

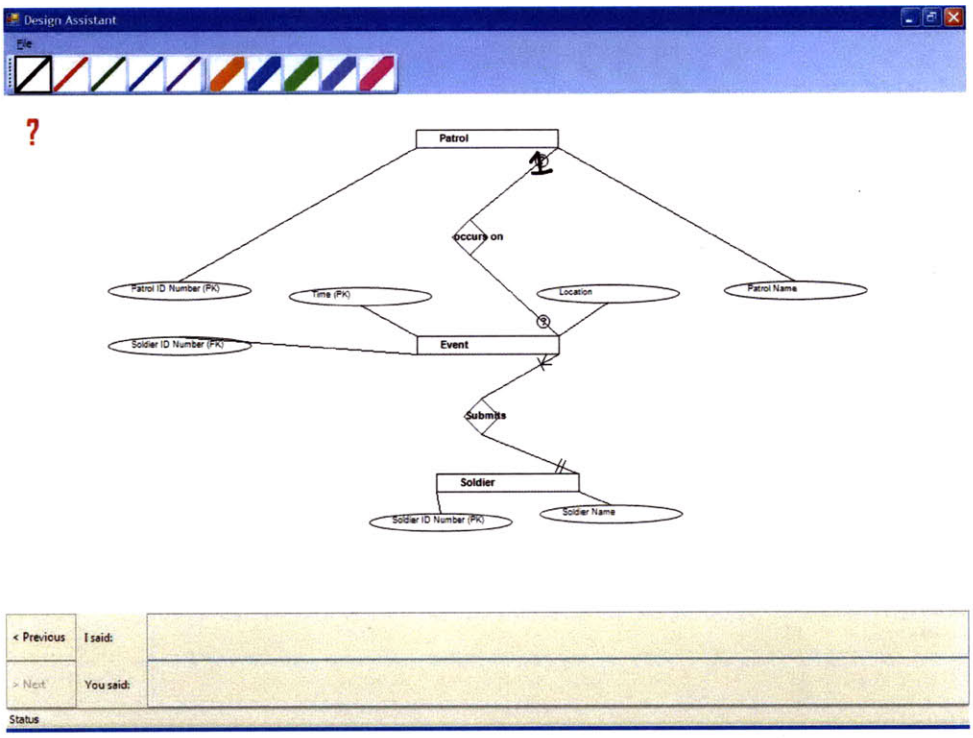


Figure 5-90: The user assign the first side of the *occurs on* relation's cardinality to be "one"

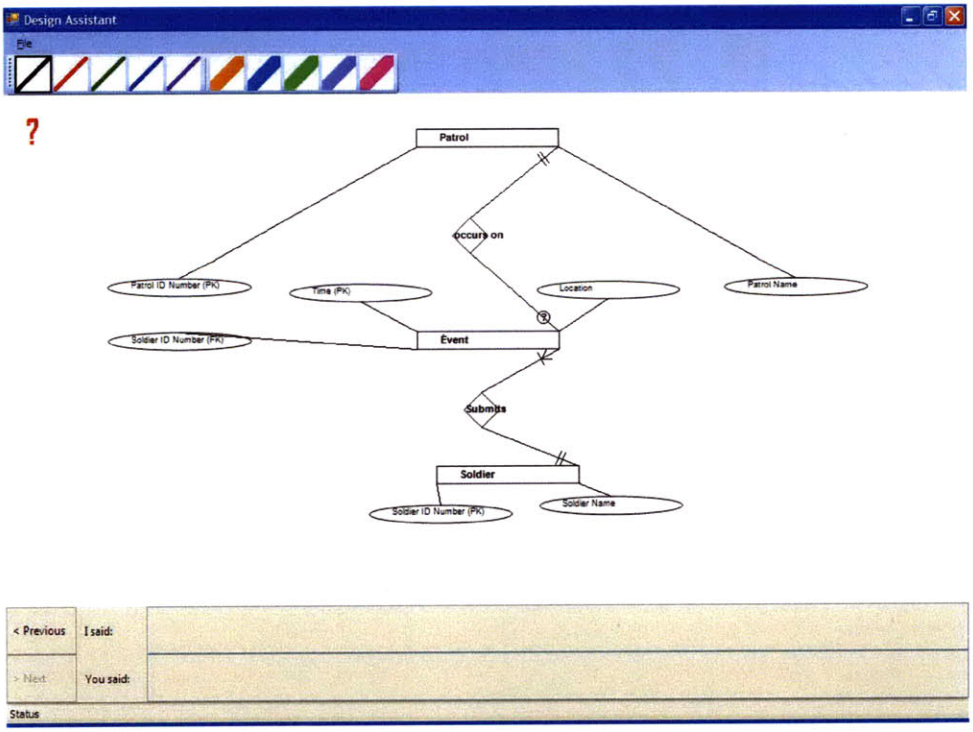


Figure 5-91: The system recognizes the user's handwriting and assigns the first side of the *occurs on* relation's cardinality to be "one"

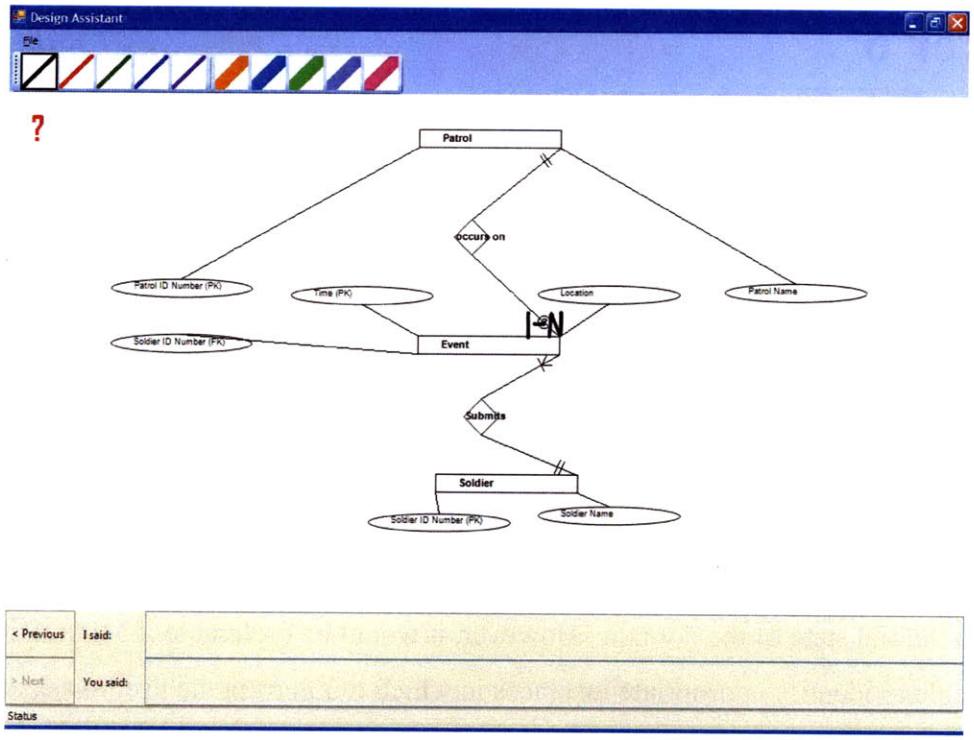


Figure 5-92: The user assign the other side of the *occurs on* relation's cardinality to be "many"

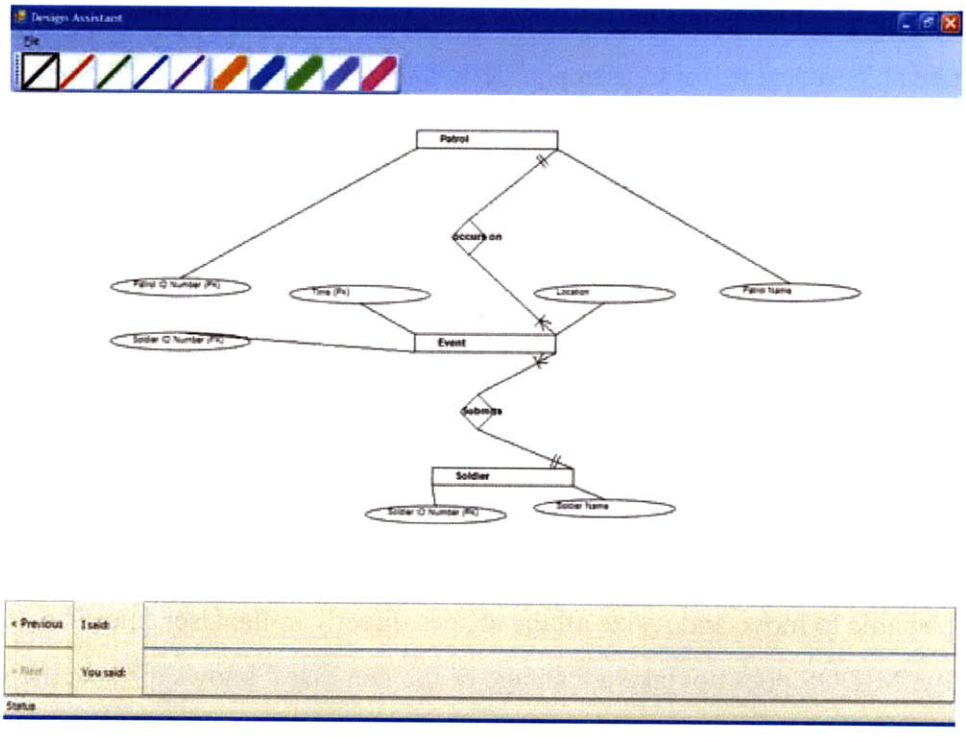


Figure 5-93: The system recognizes the user's handwriting and assigns the other side of the *occurs on* relation's cardinality to be "many".

Chapter 6

Future Work

The work described in this thesis addressed several of the original limitations of MIDOS. However, there remain many possible extensions and avenues for research with respect to the system.

In the current, modified version of MIDOS, the system will ask a question only after the user prompts it to do so. This means the user could introduce more mistakes into the design before the system has an opportunity to identify the initial problem. We chose this approach so that the system would not annoy the user with unnecessary questions, especially when the user is sketching the initial state of the domain. However, it would be preferable if MIDOS itself was eventually able to identify appropriate instances in which to interrupt the user to ask a clarifying question.

MIDOS should also be able to assist the user by filling in some aspects of the design on its own without the need to ask the user a question. This is a difficult problem because the system must be able to decide, for a particular design mistake, whether it should make a guess on its own or instead attempt to obtain the needed information from the user.

In addition, in order to build an application that utilizes the functionality of MIDOS, a developer must implement each individual *InformationRequest* by hand. However, it should be possible to generate these classes automatically from XML files, which would store all the specific domain content used by the classes (*e.g.,* multimodal question, expected input). In this manner, the domain information would be much easier to modify.

Furthermore, MIDOS currently supports some simple editing features, including the ability to draw and recognize simple shapes. Improvements in the system could therefore also be directed at enabling the user to directly manipulate the objects in the world. As an example, users should be able to move and resize all the shapes directly in the User Interface.

Finally, MIDOS does not take advantage of the fact that it knows the specific speech terms that it can expect from the user in response to a question. The speech recognizer could be supplemented by a language model that would heavily weight the words and phrases expected by the system and thereby result in better speech recognition accuracy.

Chapter 7

Contributions

We successfully extended the capabilities of Adler's MIDOS system by adding support for multiple stroke recognition, simple shape recognition, and handwriting recognition. We also created the MIDOS Interface, making MIDOS a domain-independent framework that can be utilized by any application that wishes to incorporate symmetric, multimodal interaction.

Finally, we developed a database design application that demonstrates these new features of MIDOS. This application allows a user to sketch an initial design and for the user and computer to engage in a dialogue to refine the initial design. The application uses the final design to implement and run a relational database. If the relational database is used, the application is able to make plausible inferences out of the incomplete data stored in the database in order to facilitate another dialogue between the user and computer to further refine that initial design and eliminate the redundant storage of information.

The work performed as part of this thesis brings us closer to our goal of supporting an interaction between a human and computer that is as effortless as a conversation between two people.

Bibliography

Adler, A. D. (2009). *MIDOS: Multimodal Interactive DialOgue System.* PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA.

Hammond, T., & Davis, R. (2005). LADDER, a sketching language for user interface developers. *Elsevier, Computers and Graphics 28* , 518-532.

Hammond, T., & Davis, R. (2002). Tahuti: A Geometrical Sketch Recognition System for UML Class Diagrams. *2002 AAAI Spring Symposium on Sketch Understanding* .

Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (Third Edition ed.). New York, NY: McGraw-Hill Companies, Inc.

Sezgin, T. M., Stahovich, T., & Davis, R. (2001, November). Sketch based interfaces: Early processing for sketch understanding. *The Proceedings of 2001 Perceptive User Interfaces Workshop (PUI'01)* .